UNIVERSITÉ DE MONTRÉAL

IMPROVING BUG TRIAGING USING SOFTWARE ANALYTICS

LE AN
DÉPARTEMENT DE GÉNIE INFORMATIQUE ET GÉNIE LOGICIEL
ÉCOLE POLYTECHNIQUE DE MONTRÉAL

MÉMOIRE PRÉSENTÉ EN VUE DE L'OBTENTION
DU DIPLÔME DE MAÎTRISE ÈS SCIENCES APPLIQUÉES
(GÉNIE INFORMATIQUE)
AOÛT 2015

UNIVERSITÉ DE MONTRÉAL

ÉCOLE POLYTECHNIQUE DE MONTRÉAL

Ce mémoire intitulé :

IMPROVING BUG TRIAGING USING SOFTWARE ANALYTICS

présenté par : AN Le
en vue de l'obtention du diplôme de : Maîtrise ès sciences appliquées
a été dûment accepté par le jury d'examen constitué de :

M. MERLO Ettore, Ph. D., président
M. KHOMH Foutse, Ph. D., membre et directeur de recherche
M. GUÉHÉNEUC Yann-Gaël, Doctorat, membre

DEDICATION

To my family

ACKNOWLEDGEMENTS

Firstly, I would like to express my deepest gratitude to my supervisor, Dr. Foutse Khomh, for his great guidance, encouragement and patience that he provided along my master's studies.

Secondly, I also would like to thank Dr. Bram Adams who gave me co-guidance to accomplish my first conference paper, and all members (professors and students) in the software engineering team of the Department of Computer Engineering. My studies would not have been possible without their helps. Particularly, I am grateful to Dr. Yann-Gaël Guéhéneuc, who organised and encouraged me to participate in inspirational discussions, weekly seminars, and workshops.

In addition, I would like to thank my committee members, Dr. Ettore Merlo, Dr. Yann-Gaël Guéhéneuc, and Dr. Foutse Khomh, for their valuable feedback on this thesis.

Finally, I would like to acknowledge my parents and my wife who are always supporting me with their best wishes throughout my research.

RÉSUMÉ

La correction de bogues est une activité majeure pendant le développement et maintenance de logiciels. Durant cette activité, le tri de bogues joue un rôle essentiel. Il aide les gestionnaires à allouer leurs ressources limitées et permet aux développeurs de concentrer leurs efforts plus efficacement sur les bogues à haute sévérité. Malheureusement, les techniques du tri de bogues appliquées dans beaucoup d'entreprises ne sont pas toujours efficaces et conduisent aux erreurs de classifications de bogues ou à des retards dans leurs résolutions, qui peuvent mener à la dégradation de la qualité d'un logiciel et à la déception de ses utilisateurs. Une stratégie de tri de bogues améliorée est nécessaire pour aider les gestionnaires à prendre de meilleures décisions, par exemple en accordant des degrés de priorité et sévérité appropriés aux bogues, ce qui permet aux développeurs de corriger les problèmes critiques le plus tôt possible en ignorant les problèmes futiles.

Dans ce mémoire, nous utilisons les approches analytiques pour améliorer le tri de bogues. Nous réalisons trois études empiriques. La première étude porte sur la relation entre les corrections de bogues qui ont besoin d'autres corrections ultérieures (corrections supplémentaires) et les bogues qui ont été ouverts plus d'une fois (bogues ré-ouverts). Nous observons que les bogues ré-ouverts occupent entre 21,6% et 33,8% de toutes les corrections supplémentaires. Un grand nombre de bogues ré-ouverts (de 33,0% à 57,5%) n'ont qu'une correction préalable : les bogues originaux ont été fermés prématurément. La deuxième étude concerne les bogues qui provoquent des plantages fréquents, affectant de nombreux utilisateurs. Nous avons observé que ces bogues ne reçoivent pas toujours une attention adéquate même s'ils peuvent sérieusement dégrader la qualité d'un logiciel et même la réputation de l'entreprise. Notre troisième étude concerne les *commits* qui conduisent à des plantages. Nous avons trouvé que ces commits sont souvent validés par des développeurs moins expérimentés et qu'ils contiennent plus d'additions et de suppressions de lignes de code que les autre commits.

Si les entreprises de logiciels pourraient détecter les problèmes susmentionnés pendant la phase du tri de bogues, elles pourraient augmenter l'efficacité de leur correction de bogues et la satisfaction de leurs utilisateurs, réduisant le coût de la maintenance de logiciels. En utilisant plusieurs algorithmes de régression et d'apprentissage automatique, nous avons bâti des modèles statistiques permettant de prédire respectivement des bogues ré-ouverts (avec une précision atteignant 97,0% et un rappel atteignant 65,3%), des bogues affectant un grand nombre d'utilisateurs (avec une précision atteignant 64,2% et un rappel atteignant

98.3%) et des commits induisant des plantages (avec une précision atteignant 61,4% et un rappel atteignant 95,0%). Les entreprises de logiciels peuvent appliquer nos modèles afin d'améliorer leur stratégie de tri de bogues, éviter les erreurs de classification de bogues et réduire la insatisfaction des utilisateurs due aux plantages.

ABSTRACT

Bug fixing has become a major activity in software development and maintenance. In this process, bug triaging plays an important role. It assists software managers in the allocation of their limited resources and allow developers to focus their efforts more efficiently to solve defects with high severity. Current bug triaging techniques applied in many software organisations may lead to misclassification of bugs, thus delay in bug resolution; resulting in degradation of software quality and users' frustration. An improved bug triaging strategy would help software managers make better decisions by assigning the right priority and severity to bugs, allowing developers to address critical bugs as soon as possible and ignore the trivial ones.

In this thesis, we leverage analytic approaches to conduct three empirical studies aimed at improving bug triaging techniques. The first study investigates the relation between bug fixes that need supplementary fixes and bugs that have been re-opened. We found that re-opened bugs account from 21.6% to 33.8% of all supplementary bug fixes. A considerable number of re-opened bugs (from 33.0% to 57.5%) had only one commit associated: their original bug reports were prematurely closed. The second study focuses on bugs that yield frequent crashes and impact large numbers of users. We found that these bugs were not prioritised by software managers albeit they can seriously decrease user-perceived quality and even the reputation of a software organisation. Our third study examines commits that lead to crashes. We found that these commits are often submitted by less experienced developers and that they contain more addition and deletion of lines of code than other commits.

If software organisations can detect the aforementioned problems early on in the bug triaging phase, they can effectively increase their development productivity and users' satisfaction, while decreasing software maintenance overhead. By using multiple regression and machine learning algorithms, we built statistical models to predict re-opened bugs among bugs that required supplementary bug fixes (with a precision up to 97.0% and a recall up to 65.3%), bugs with high crashing impact (with a precision up to 64.2% and a recall up to 98.3%), and commits inducing future crashes (with a precision up to 61.4% and a recall up to 95.0%). Software organisations can apply our proposed models to improve their bug triaging strategy by assigning bugs to the right developers, avoiding misclassification of bugs, reducing the negative impact of crash-related bugs, and addressing fault-prone code early on before they impact a large user base.

CO-AUTHORSHIP

Earlier studies in the thesis were published/submitted as follows:

- Supplementary Bug Fixes vs. Re-opened Bugs
  Le An, Foutse Khomh and Bram Adams, in *Proceedings of the 14th IEEE International Working Conference on Software Code Analysis and Manipulation (SCAM), 28-29 September, 2014.*
  My contribution: data mining and analysis, paper writing, and presentation at the conference.

- Challenges and Issues of Mining Crash Reports
  Le An and Foutse Khomh, in *Proceedings of the 1st International Workshop on Software Analytics (SWAN), 2 March, 2015.*
  My contribution: design of research plan, paper writing, and presentation at the workshop.

- An Empirical Study of Highly-impactful Bugs in Mozilla Projects
  Le An and Foutse Khomh, in *Proceedings of the 2015 IEEE International Conference on Software Quality, Reliability and Security (QRS), 3-5 August, 2015.*
  My contribution: data mining and analysis, paper writing, and presentation at the conference.

- An Empirical Study of Crash-inducing Commits in Mozilla Firefox
  Le An and Foutse Khomh, in *Proceedings of the 11th International Conference on Predictive Models and Data Analytics in Software Engineering (PROMISE), 21 October, 2015.*
  My contribution: data mining and analysis, paper writing.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1 INTRODUCTION

During software development and maintenance, debugging plays an important role, especially after the first release of a software system. According to a report by the US Department of Commerce [1], bug fixing activities account for up to 80% of software development overhead. Bug fixing efficiency would decide the productivity and users' satisfaction for a software organisation. Among all of the detected defects, the most important ones will be filed into bug tracking systems (*e.g.*, Bugzilla[1] and Jira[2]), where bugs will generally experience the following statuses: opened, new, assigned, resolved, verified, and closed [2].

Bugs usually have different impacts on a software system and different fixing difficulties for developers. When a bug is newly opened, software managers will prioritise the bug and assign it to developers to fix. This process is called *bug triaging*, which can help development teams focus their limited resources to resolve the most impactful defects and avoid complaints from end users. In this thesis, we study bug triaging strategy from the following three aspects: the relation between re-opened bugs and supplementary bug fixes, bugs that lead to frequent crashes and impact a large number of users, and commits that induce crashes. The study of each aspect would help software managers improve their bug triaging strategy; increasing developers' productivity and users' satisfaction, while reducing software maintenance overhead.

1.1 Relation between Supplementary Fixes and Re-opened Bugs

A typical bug fixing cycle involves the reporting of a bug, the triaging of the report, the production and verification of a fix, and the closing of the bug. However, previous work has studied two phenomena where more than one fix are associated with the same bug report. The first one is the case where developers re-open a previously fixed bug in the bug repository (sometimes even multiple times) to provide a new improved bug fix that replace a previous fix, which is called a *re-opened bug* [2]. Shihab et al. [2] argued that re-opened bugs increase maintenance overhead, degrade the overall user-perceived quality of the software, and lead to repetitive work by already busy developers. The second one is the case where multiple commits in the version control system contribute to the same bug report, which is called *supplementary bug fixes* [3]. According to a manual investigation by Park et al. [3], supplementary bug fixes may be due to missed porting changes, incorrect handling

[1]https://www.bugzilla.org
[2]https://www.atlassian.com/software/jira

of conditional statements, or incomplete refactorings. Even though both phenomena seem related, they have never been studied together, *i.e.*, are supplementary fixes a subset of re-opened bugs or the other way around? In this research, we investigate the interplay between both phenomena in five open-source software projects: Mozilla, Netbeans, Eclipse JDT Core, Eclipse Platform SWT, and WebKit. We find that re-opened bugs account for between 21.6% and 33.8% of all supplementary bug fixes. However, 33.0% to 57.5% of re-opened bugs had only one commit associated; meaning that the original bug reports were prematurely closed instead of being fixed correctly. We build predictive models for re-opened bugs using historical information about supplementary bug fixes with a precision between 72.2% and 97.0%, as well as a recall between 47.7% and 65.3%. Software organisations can use our proposed models to improve their bug triaging strategy to reduce the misclassifications of bugs.

1.2 Bugs that Crashed Frequently and Impact a Large User Base

Nowadays, crash reporting tools are embedded in many software systems to collect information about crashes in the field (*i.e.*, when a program stops functioning properly in a user environment). Crashes with the same crashing signature, the stack trace of the failing thread, will be grouped automatically into a *crash type*. Usually, software quality managers prioritise crash types by the number of crash occurrences, then file the top crash types into bug reports. Crash reports provide useful reference to analyse and resolve the crashing bugs. They could help software practitioners locate erroneous code, understand the impact of failures, and prioritise crash-related bug reports. In a previous study, Khomh et al. [4] proposed an entropy-based crash triaging approach that can help software organisations identify crash-types that affect a large user base with high frequency. We refer to bugs associated to these crash-types as *highly-impactful bugs*. The proposed triaging approach can identify highly-impactful bugs only after they have led to crashes in the field for a period of time. Therefore, to reduce the impact of highly-impactful bugs on user-perceived quality, an early identification of these bugs is necessary. In this thesis, we examine the characteristics of highly-impactful bugs in Mozilla Firefox and Fennec for Android and propose statistical models to help software organisations predict them early on before they impact a large population of users. Results show that our proposed prediction models can achieve a precision up to 64.2% (in Firefox) and a recall up to 98.3% (in Fennec). We also evaluate the benefits of our proposed models and found that, on average, they could help reduce 23.0% of Firefox's crashes and 13.4% of Fennec's crashes, while reducing 28.6% of impacted machine profiles for Firefox and 49.4% for Fennec. Software organisations could use our prediction models to catch highly-impactful bugs early during the triaging process, preventing them from impacting a larger user base.

1.3 Commits that Lead to Crashes

Although the approaches focusing on triaging crash-related bugs can help software practitioners increase their debugging efficiency on crashes, these techniques can only be applied after the crashes occurred and already affected a large population of users. To help software organisations detect and address crash-prone code even earlier, we conduct another case study on commits that would lead to crashes, called *crash-inducing commits*, in Mozilla Firefox. We found that crash-inducing commits are often submitted by developers with less experience. Also, developers perform more addition and deletion of lines of code in crash-inducing commits. We built predictive models to help software organisations detect and fix crash-prone bugs once a defective commit is integrated into the version control system. Our predictive models achieve a precision of 61.4% and a recall of 95.0%. Software organisations can use our proposed predictive models to track and fix crash-prone commits as soon as possible before they negatively impact users; increasing bug fixing efficiency and user-perceived quality.

1.4 Research Statement

Prior research studied either supplementary bug fixes or re-opened bugs, but researchers have never linked the two phenomena together. In addition, in current software organisations, most crash collecting systems triage crashes and crash-related bugs merely by the corresponding crash frequency. In this thesis, we study the relationship between supplementary bug fixes and re-opened bugs and propose an entropy-based bug triaging approach as well as a crash-prone commit detection approach to improve bug triaging techniques aimed at increasing development efficiency and users' satisfaction.

1.5 Thesis Overview

- *Does every re-opened bug need supplementary fixes (Chapter 3)?*
  We extract bugs that have been re-opened and bugs that require multiple patches from five open-source software systems to investigate their relationship. We propose statistical models to predict bug re-opening in supplementary fixes to help software practitioners avoid misclassification of bug reports.
- *Were the bugs with high impact on end users prioritised (Chapter 4)?*
  We mine crash reports and bug reports of Mozilla Firefox and Fennec for Android and found that the distribution of bugs in the user base has not been taken into account. We propose statistical models to predict bugs with both high crash frequency and large

impact on users. In addition, we analyse the gain and loss that can be achieved by applying our proposed models.

- *How can we detect crash-prone commits (Chapter 5)?*
  We mine crash reports and commit logs of Mozilla Firefox to identify crash-inducing commits. After investigating the characteristics of the crash-inducing commits, we build statistical models to predict them.

1.6 Thesis Contribution

In this thesis, we carry out empirical studies on different aspects of bug triaging techniques. Our contributions are as follows:

- We found nearly 50% of bugs that have been re-opened only have one related bug fix. It implies that these bugs have been prematurely closed and that developers who closed the bugs may have negative attitude towards them. We propose predictive models to help software organisations prevent potential re-opened bugs in supplementary bug fixes to increase their development productivity and the overall user-perceived quality.
- We propose an entropy-based bug triaging approach and predictive models to assist software managers and practitioners detect bugs that may lead to high crash frequency and affect a large user base early on during the triaging process to prevent them from continuously impacting end users.
- We found that crash-inducing commits are often submitted by less experienced developers. They contain more addition and deletion of lines of code than other commits. We propose statistical models to predict crash-inducing commits. These models can help software managers identify crash-prone code before they cause negative impact to the project and end users.

1.7 Organisation of the Thesis

The rest of this thesis is organised as follows:

- Chapter 2 outlines literature review in the areas of bug triaging, supplementary bug fixes, bug re-opening, crash report analysis, entropy analysis, mining software repositories, and fault prediction.
- Chapter 3 presents our empirical study on supplementary bug fixes and re-opened bugs, as well as their relationship.

- Chapter 4 presents our empirical study on highly-impactful bugs, *i.e.*, bugs that cause a large number crash occurrences and affect a lot of users in a software system.
- Chapter 5 presents our empirical study on commits that will induce subsequent crashes.
- Chapter 6 summarises and conclude the thesis and discuss future work.

CHAPTER 2 LITERATURE REVIEW

2.1 Bug Triaging

Bug triaging is the process that consists in screening and prioritising bugs to allow software organisations to focus their limited resources on bugs with high impact on software quality. In many software organisations, quality managers use automatic bug triaging systems to decide bugs' priority and assign bugs to the appropriate debuggers. The accuracy of the bug triaging systems will affect the debugging efficiency, the quality of the software, and even the satisfaction of end users.

In previous studies, researchers proposed different defect triaging techniques to help software organisations improve their triaging activities. Anvik et al. [5] introduced a semi-automated approach to ease the assignment of bug reports to a developer. They applied a supervised machine learning algorithm to learn the kinds of bug reports resolved by each developer in the bug repository, then to suggest a small number of suitable developers to resolve each new bug. Canfora and Cerulo [6] also proposed a semi-automatic approach to select the best candidate set of developers to resolve new change requests. This approach identifies candidate developers using the textual description of the change requests. Menzies and Marcus [7] proposed an automated approach, SEVERIS, to help triage teams assign severity levels to bug reports. Their approach is based on text mining and machine learning techniques applied to existing sets of bug reports. Weiss et al. [8] proposed an approach to help triage teams automatically predict the fixing effort (*i.e.*, bug fixing time) of a bug based on the average fixing time of its similar and earlier bugs. This approach allows for early effort estimation to help triage teams better assign issues. Jeong et al. [9] studied bug tossing (*i.e.*, reassignment of bug reports) and found that tossing bugs lead to longer bug fixing time. They proposed a tossing graph model, which captures past tossing history, to reduce tossing steps and improve the accuracy of previous automatic bug assignment approaches. Khomh et al. [4] proposed an entropy-based technique to triage crash-types in Firefox. Their proposed approach achieves a better classification of crash-types than the current technique applied by Firefox teams.

In this research, we study bug triaging technique on three aspects: supplementary bug fixes and re-opened bugs, crash-related bugs, and crash-inducing commits. We apply statistical analyses to help software organisations improve their current bug triaging systems to augment their development productivity and increase users' satisfaction.

2.2 Supplementary Bug Fixes

Park et al. defined supplementary bugs fixes as "patches that were later applied to supplement or correct initial fix attempts" [3]. They studied supplementary fixes on three open-source systems by a manual inspection. They found that supplementary fixes may be caused by several reasons, *e.g.*, missed porting changes, incorrect handling of conditional statements, incomplete refactorings, etc. Also, they discovered that only a very small portion of supplementary fixes have a content similar to their initial bug fixes and that only 14% to 15% of files in the studied supplementary fixes overlap with the initial fix locations nor had direct structural dependencies on them.

Yin et al. [10] studied incorrect bug fixes in large operating systems. They found that 14.8% to 24.4% of studied fixes for post-release bugs are incorrect and that concurrency bugs are the most difficult to fix. Their results show that a considerable portion of bugs were fixed more than once, *i.e.*, they needed supplementary fixes.

By analysing small changes in Lucent 5ESS, Purushothaman et al. [11] found nearly 40% of bug fixes having introduced other defects. In other words, these bug fixes need subsequent fixes to address their defects.

2.3 Bug Re-opening

In a bug tracking system, a closed bug may be opened again by developers due to various reasons, such as inefficiency of previous bug fixes or lack of clarity in the reproducibility of the bug before closure. Researchers and software practitioners tend to consider re-opened bugs primarily as a negative factor because repeated work increases maintenance overhead, degrading the software quality.

Shihab et al. [2] discussed the risk of re-opened bugs and built prediction models to prevent bug re-opening. Zimmermann et al. [12] characterised how bug reports are re-opened by analysing the Microsoft Windows operating system project in an empirical study. They used a mixed-methods approach, categorising the reasons for re-opening based on a survey of 358 Microsoft employees, then running a quantitative study of Windows bug reports by focusing on factors related to bug report edits and relationships between people involved in handling the bugs. They built statistical models to describe the impact of various metrics on re-opened bugs ranging from the reputation of the opener to how the bug was found. Xia et al. [13] evaluate the effectiveness among 10 supervised learning algorithms to predict the re-opening probability of a bug report. They found that Bagging and Decision Table (IDTM) achieve the best performance. Gu et al. [14] studied bad fixes, namely bug fixes

failing to fix a bug or creating new bugs, in three open-source systems. They found that 38% to 50% of re-opened bugs are due to bad fixes, which block other bugs or are duplicated by other bugs in Bugzilla. Non-reproducible decision is an important reason behind bug re-opening. Joorabchi et al. [15] investigated one industrial and five open-source bug databases and found that non-reproducible bug reports account for 17% of all bug reports and that they remain active longer than other bugs. Those bugs can be mainly classified as "Interbug Dependencies" and 66% of *Fixed* non-reproducible reports were actually reproduced and fixed. In other words, these bug reports were misclassified. However, none of these studied has considered the relation between supplementary bug fixes and bug re-openings, neither have they tried to investigate whether all re-opened bugs need supplementary fixes.

2.4 Crash Report Analysis

Podgurski et al. [16] introduced an automated failure clustering approach for the classification of crash reports to facilitate their prioritisation and the diagnostic of their root causes. By mining crash reports in Mozilla Firefox, Khomh et al. [4] proposed an entropy-based approach that can be used to identify crash types by their crashing frequency and their crashing dispersion among users. Inspired by the work of Khomh et al., Wang et al. [17] analysed crash information in Firefox and Eclipse and proposed an algorithm that can be used to locate and rank buggy files as well as a method to identify duplicate and related bug reports. Kim et al. [18] studied crash reports and impacted source files in Firefox and Thunderbird to predict top crashes before a new release of a software system.

Most of these researchers used Mozilla Socorro crash reporting system [19] as a subject system. Because, as of the writing of this thesis, only the Mozilla Foundation has opened its crash reports to the public [17]. Though Wang et al. [17] studied another system, Eclipse, they could obtain crash information only through the stack traces contained in the bug reports (instead of using crash reports). However, stack trace information is not always available in bug reports for the majority of software systems. Dang et al. [20] proposed a method, ReBucket, to improve the current crash reports' clustering technique based on call stack matching. The subject crash collecting database, Microsoft's Windows Error Reporting (WER) system, is not accessible for every researcher.

In this thesis, we study crash reports from Mozilla Socorro system to investigate whether the current crash-related bugs are accurately triaged and whether we can propose a better bug triaging approach to software organisations.

2.5 Entropy Analysis

Entropy metrics capture the dispersion of information. Entropy analyses are extensively applied in software engineering studies. Bianchi et al. [21] measured the entropy of a software system to assess its degradation. They developed a tool based on software representation models that can automatically compute entropy metrics before and after every maintenance intervention. By treating a software system as an information source, Hafiz et al. [22] used different entropy measures, Shannon, Hartley, and Renyi entropy, to extract different types of information from the system. They found that files that are more functional and descriptive provide a larger amount of entropy. Examining the complexity and chaos associated with the development process, Hassan et al. [23] used the Shannon entropy to measure the complexity of systems. They concluded that entropy reflects the code complexity and the possibility of decay of a software system. Zaman et al. [24] also applied the normalised Shannon entropy to compare security and performance bugs in terms of bug fixing patches to assess the complexity of bug fixes. Kim et al. [25] introduced new software complexity metrics (*i.e.*, class complexity and inter-object complexity) for object-oriented software systems based on Shannon entropy. Chapin et al. [26] analysed entropy metrics of software systems and concluded that software practitioners can gather a good insight on how the maintenance of the software system should be performed by observing any abrupt change in the entropy of a software system. Using the Shannon entropy measure to quantify information content in databases, Unger et al. [27] introduced a measure of the general vulnerability of databases based on entropy values and proposed a technique based on entropy analysis. Khomh et al. [4] proposed an entropy-based approach that can be used to identify crash-types with both high frequency and high distribution in a users' base. Although entropy helps determine which problems are more frequently reported than others, it is always difficult to determine whether certain bugs are redundant or affect the entire user population. Anvik et al. [28] provided an alternate way to process an open bug repository and eliminating bugs that are duplicates or irrelevant.

In this research, we intend to study whether entropy can help software practitioners to classify bug reports more accurately than their current techniques, and, if yes, to which extent practitioners can save time and increase users' satisfaction.

2.6 Mining Software Repositories

The study on mining software repositories (MSR) has become popular in the last decade. It assists software researchers in discovering interesting and actionable information in software systems [29, 30] by analysing data on a variety of software repositories, such as bug database

(e.g., Bugzilla or Jira) or version control systems (e.g., Git or Mercurial). Hassan [30] indicated that "MSR researchers have proposed techniques that augment traditional software engineering data, techniques and tools to solve important and challenging problems, such as identifying bugs, and reusing code, which practitioners must face and solve on a daily basis". This technique will allow us better understand the reason of bugs in the subject systems and propose improved bug triaging strategies to help software organisations augment their development productivity and increase the user-perceived quality of their product.

2.7 Defect Prediction

Flaws are not favourable to software development. It is worth building predictive models to understand the reasons of the faults and prevent potential faults early on before they appear in the filed.

2.7.1 Traditional Defect Prediction Techniques

Traditional defect prediction techniques used coarse-grained metrics, such as bug report metrics, to identify defect-prone modules or specific types of bugs. By using social factors, technical factors, coordination factors, and prior-certifications factors, Hassan et al. [31] created decision trees to predict ahead of time the certification result of a build for a large software project at IBM Toronto Lab. Shihab et al. [2] extracted metrics from bug reports and built models using C4.5, Zero-R, Naive Bayes, and Logistic Regression algorithms to predict bug re-opening in three open-source projects. As a complementary work, Zimmermann et al. [12] used Logistic Regression models to predict bug re-opening in Windows.

2.7.2 Just-in-Time Defect Prediction Techniques

Though traditional defect prediction techniques can help software organisations prevent defects to some extent, developers can only identify error-prone modules responsible for these defects after the defects have been filed into bug reports. During the period between the integration of the defective code into the version control system and the opening of the bug report, a defective commit could negatively impact a large user base.

Just-in-Time defect prediction techniques are designed to predict defects at commit level; allowing developers to find and fix defects once a commit is submitted for integration in version control systems. Kamei et al. [32] used a wide range of source code metrics to predict defect-prone commits in six open-source systems and five commercial systems. Fukushima et al. [33] applied Just-in-Time defect prediction techniques to cross-project defect predictions

and found it to be viable for projects with little historical data. Using a number of code and process factors extracted at change level, Misirli et al. [34] built statistical models to predict high impact fix-inducing changes.

In this work, we apply different machine learning algorithms to predict development faults, such as bug re-opening in supplementary bug fixes, bugs that impact a large population of users (traditional defect prediction techniques), as well as commits that induce future crashes (Just-in-Time defect prediction techniques). By comparing the performance of these predictive algorithms, we propose the best predictive models to allow software organisation to classify bug reports more efficiently and prevent future flaws. Using the predictive models, software practitioners can also understand which factors of their development process lead to more faults. With the proposed prediction models, they can also focus on reviewing fault-prone commits.

2.8 Chapter Summary

In this chapter, we briefly introduce the importance of bug triaging process and previous studies in this field. In a bug database, there exist different types of bugs. We specially review the related work on supplementary bug fixes, re-opened bugs, and crash-related bugs by discussing the analytic approaches applied in previous studies, such as mining software repositories, crash report analysis, entropy analysis, and fault prediction techniques.

In the following chapters, we describe our case studies on the aforementioned bug types to assess current bug triaging techniques used in software organisations and propose improved approaches to help software practitioners concentrate on the most important bugs to increase development productivity and users' satisfaction.

CHAPTER 3 SUPPLEMENTARY BUG FIXES VS. RE-OPENED BUGS

Bug fixing is a major activity during software development process. A typical bug fixing cycle includes many different phases, performed by a variety of stakeholders: reporting of the bug (users), production of a fix (developers), verification of the fix (testers), and closing of the bug definitively (managers). For some bugs, developers have to try multiple times before fixing a bug. As a result of these several attempts, bug reports are sometimes re-opened, which may incur longer fixing time [2] and hence is likely to degrade users' satisfaction and decrease the productivity of development teams, as developers must rework the same bug multiple times: re-analysing the context of the bug, reading previous discussions about the bug and examining previous failed fixes (proposed for the bug). Thus, it is important to identify flawed bug fixes early before they can crash in the field.

Work on failed bug fixes has focused on two areas, *i.e.*, supplementary bug fixes and re-opened bugs. Supplementary bug fixes correspond to multiple commits linked (via their commit log message) to the same bug report. Park et al. [3] investigated supplementary fixes in three open-source projects: Eclipse JDT core, Eclipse SWT, and Mozilla. They concluded that supplementary fixes are typically caused by forgetting to port changes, by incorrect handling of conditional statements, or by incomplete refactorings. Shihab et al. [2], Zimmermann et al. [35], and Xia et al. [13] proposed models for the prediction of re-opened bugs. Although both areas obviously are related and have spawned two active research communities, their exact relation has never been studied: are re-opened bugs a subset of supplementary bug fixes (or the other way around)?

This chapter analyses the relation between supplementary bug fixes and re-opened bugs by studying the factors that indicate whether a bug fix will require supplementary fixes and–or will be re-opened. Knowing the characteristics of fixes that require supplementary fixes will help to better focus code review activities and prevent known bugs from re-appearing in the field. Knowing the characteristics of bugs that must be re-opened will help to predict the probability of bug re-opening to reduce the maintenance overhead and improve the overall quality of software. Using bug fix and bug re-opening information from five open-source software projects, Mozilla, Netbeans, Eclipse JDT Core, Eclipse Platform SWT, and WebKit, we address the following three research questions:

RQ1: *What is the proportion of bugs among all bug reports that require supplementary bug fixes or are re-opened?*

This research question replicates the work of Park et al. [3], who analysed Eclipse JDT core, Eclipse SWT, and Mozilla and found that between 22.5% to 32.8% of resolved bugs involved more than one fixing attempt. With this question, we want to verify whether supplementary fixes are related to frequent failure and, hence, whether they are worth investigating in details. We find that the proportion of bugs that required supplementary bug fixes in Mozilla[1], Netbeans[2], Eclipse JDT Core[3], Eclipse Platform SWT[4] and WebKit[5] accounts for, respectively, 23.8%, 17.2%, 26.9%, 25.9% and 10.3% of the total number of resolved bugs reports. Only the results for Webkit are not similar to those of Park et al. We attribute the difference to the style of commit messages in this project where many commits cannot be mapped to their corresponding bug reports.

RQ2: *What is the relation between supplementary bug fixes and re-opened bugs?*

We want to understand whether bug fix failures are caught early during reviews and testing activities or whether they slip through these verification and validations steps and crash in the field, prompting the re-opening of bug reports. According to our results, between 21.6% and 33.8% of supplementary fixes have been re-opened at least once. In addition, bug re-openings tend to coincide with multiple fixing attempts, long fixing period, and multiple developers. Surprisingly, we also found that, contrary to intuition, 33.0% to 57.5% of the re-opened bugs were not detected as supplementary fixes, instead they are mostly due to premature closing of bugs.

RQ3: *Can we predict the re-opening of supplementary bug fixes?*

Re-opened bugs may increase maintenance costs, degrade the overall software quality, and users' satisfaction [2]. In this research question, we use GLM, C5.0, ctree, cforest, and randomForest [36] algorithms with attributes about developers' working habits, commit logs, bug fix, and development teams' dynamic to build models that can predict whether or not a bug that required supplementary fixes before initial closing of its report will be re-opened. Our models can correctly predict whether or not a bug requiring supplementary fixes will need to be re-opened, with a precision between 72.2% and 97.0% and a recall between 47.7% and 65.3%. Software organisations could use our proposed models to predict potential failures of their bug fixes and the re-opening of bug reports, hence preventing these bugs from re-appearing in the field.

[1] `http://www.mozilla.org/`
[2] `https://netbeans.org/`
[3] `http://www.eclipse.org/jdt/core/`
[4] `http://www.eclipse.org/swt/`
[5] `https://www.webkit.org`

Chapter Overview

Section 3.1 describes the design of our case study. Section 3.2 describes and discusses the results of our three research questions. Section 3.3 discusses the results of our replication study in the context of previous work. Section 3.4 discloses the threats to validity of our study. Section 3.5 summarises this chapter.

3.1 Study Design

This section presents the design of our case study, which aims to address the following three research questions:

RQ1: What is the proportion of bugs among all bug reports that require supplementary bug fixes or are re-opened?

RQ2: What is the relation between supplementary bug fixes and re-opened bugs?

RQ3: Can we predict the re-opening of supplementary bug fixes?

3.1.1 Data Collection

Since our study replicates existing work on supplementary bug fixes [3] and re-opened bugs [2], we selected the following five open-source software projects: Mozilla, Netbeans, Eclipse JDT Core, Eclipse Platform SWT, and WebKit. Mozilla, which was also used by Park et al. [3], is a Web project that includes several sub-products, such as the Firefox Internet browser and the Thunderbird e-mail client. Eclipse, which was used by both Park et al. and Shihab et al. [2], is an integrated development environment (IDE) supporting various programming languages. In addition, to compare with the results in [2] and [3], we introduced two other projects: Netbeans and WebKit. Similar to Eclipse, Netbeans is another commonly used IDE. WebKit is a layout engine software component for rendering Web pages that powers Apple's safari browser.[6]

3.1.2 Data Processing

Figure 3.1 shows an overview of our analysis approach. First, we extract bug fix information from version control systems (*i.e.*, Mercurial and Git) and apply the algorithm of Park et al. to identify supplementary bug fixes [3]. Then, we mine the bug repositories (*i.e.*, Bugzilla) of our five subject projects to identify re-opened bugs. Using these data, we compute several

[6] All our studied data repositories, and analysis scripts are available here:
`https://github.com/anlepoly/supplementary_fixes`

metrics and build statistical models to predict the re-opening probability of bugs that require supplementary fixes. The remainder of this section elaborates on each of these steps.

Identification of Bug Fixes

We extract the revision history of each subject project from the *Mercurial* (for Mozilla and Netbeans) and *Git* (for Eclipse and WebKit) repositories. We obtained the data of the three repositories Mozilla, Netbeans, and Eclipse from the MSR 2011 challenge, which respectively cover the period from March 2007 to August 2010, from January 1999 to June 2010, and from October 2001 to June 2010. The WebKit data cover the period from August 2001 to June 2014.

Next, we parse the files' revision logs to extract the following commit information: revision numbers, committer names, commit dates, commit messages, number of changed files, and number of inserted/deleted lines. We apply heuristics from Fischer et al. [37] to identify bug fixing commits. More specifically, we apply the following regular expressions incrementally to match bug report identifiers:

```
(bug|issue)[:#\s_]*[0-9]+
(b=|#)[0-9]+
[0-9]+\b
\b[0-9]+
```

Finally, we cross-check the bug IDs obtained from commit logs with the Bugzilla repository to ensure that they represent actual bug reports, *i.e.*, check whether the extracted bug IDs exist in the corresponding Bugzilla repository.

Identification of Supplementary Bug Fixes

We apply the algorithm proposed by Park et al. [3] to track supplementary bug fixes. This algorithm considers, as a supplementary bug fix, any fix where the commit message contains the bug ID of a previous bug-fixing commit. Therefore, among all detected bug-fixing commits, we search for revisions where the bug ID is repeated. During this process, we observed that in some commit messages, committers just mentioned the revision number of a previous bug fix instead of the bug ID. Hence, we enhance Park et al.'s heuristic by also matching these revisions to the corresponding bugs.

Table 3.1 presents an example of supplementary bug fixes. In this table, there are three revisions that mention the same bug ID #462381. Revision 21149 is the initial bug fix, while revisions 34890 and 34902 are supplementary bug fixes.

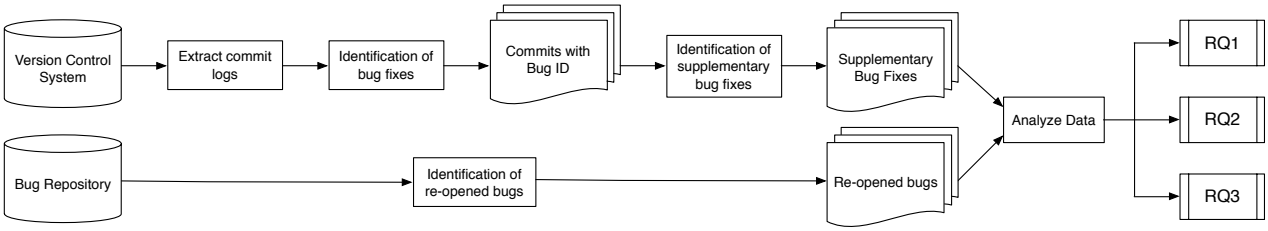

Figure 3.1 Overview of our approach to study the relation between supplementary fixes and re-opened bugs

Table 3.1 Supplementary bug fixes of bug #462381

changeset	21149:7aeaf064ad9f
date	Fri Oct 31 09:07:15 2008 -0700
summary	**Bug 462381** - Build layout directories in parallel r=ted sr=roc
churn	12 files changed, 16 insertions(+), 464 deletions(-)
... ...	
changeset	34890:fae81b8a5648
date	Fri Nov 13 14:40:00 2009 -0500
summary	**bug 462381** - sprinkle magic PARALLEL_DIRS fairy dust about the build system r=ted.mielczarek
churn	12 files changed, 191 insertions(+), 173 deletions(-)
... ...	
changeset	34902:827d8651799e
date	Mon Nov 16 07:57:15 2009 -0500
summary	bustage fix from **bug 462381**
churn	1 files changed, 4 insertions(+), 2 deletions(-)

After the identification of supplementary bug fixes, we organise all bug fixes into two groups (similarly to [3]):

- *Type I bug fix* - bug fixes that definitively solve the bug in the first attempt (*i.e.*, no supplementary fix is needed)
- *Type II bug fix* - bug fixes that require supplementary fixes before the bug can be solved.

Identification of Re-opened Bugs

In Bugzilla, a re-opened bug may be marked "REOPENED" in two places: in the "status" field, when it is currently re-opened and not yet solved and in its "history" list, if it was once re-opened but afterwards the status had been changed to something else (*e.g.*, again "CLOSED"). Instead of just looking at the final status of a bug, we check the bug's "history" list and find whether there is at lease one "REOPENED" tag. In the case of Mozilla, Netbeans, Eclipse JDT Core, and Eclipse Platform SWT, we extract this information directly from the Bug SQL databases that were provided for MSR 2011 Mining Challenge. In the

case of WebKit, we concatenate the Bugzilla URL with each detected bug ID to download the "history" page of the bug. Then, we check whether the tag "REOPENED" exists in the bug's history. For example, to check whether bug #32698 in WebKit was once re-opened, we combine the history link of *WebKit Bugzilla* and the target bug ID as follows:

`https://bugs.webkit.org/show_bug.cgi?id=32698`

3.2 Case Study Results

This section presents and discusses the results of our three research questions.

RQ1: What is the proportion of bugs among all bug reports that require supplementary bug fixes or are re-opened?

Motivation. This question is preliminary to the other questions. It provides quantitative data on the proportion of bugs that required supplementary bug fixes and bugs that have been re-opened in our five subject systems. In this research question, as in the study of Park et al. [3], we determine whether bug fixes fail frequently, how fast the bugs are fixed for good, and how many developers are needed to fix the bugs. These results will clarify the prevalence (and hence importance) of supplementary bug fixes, and allow us to compare our findings with those from [3].

Approach. We identify supplementary bug fixes by classifying bug fixes from the five systems into two categories: Type I and Type II bug fixes, as discussed in Section 3.1.2. We identify re-opened bugs following the heuristics described in Section 3.1.2, and compute the proportion of bug reports that have been re-opened. For each bug report, we also compute the number of fixing attempts required for the bug, the duration (in days) of the fixing period, and the number of developers that contributed to finally fix the bug. Because Type II bugs contain multiple fixes, we respectively calculate their number of bug fixes and number of bug reports (*i.e.*, all fixes corresponding to the same bug ID count for one).

Findings. Overall, in the five studied projects, Type II bug reports account for 10.3% to 26.9% of all the bug reports. Table 3.2 shows descriptive statistics about our subject systems.

Although Netbeans has the highest number of commits, it has the lowest percentage of commits that fix bugs. Further manual analysis shows that more than 20% of the commit messages only mentioned the product repository links instead of bug IDs (*e.g.*, "Automated merge with `http://hg.netbeans.org/cnd-main/`"). These commits cannot be identified as fixing bugs. There are also many very short commit messages from which we can not extract any useful information about bug with the heuristic introduced in Section 3.1.2. This result

Table 3.2 Descriptive statistics of the subject systems

	Mozilla	**Netbeans**	**JDT Core**	**Platform SWT**	**WebKit**
Studied period	3/2007 - 8/2010	1/1999 - 6/2010	10/2001 - 6/2010	10/2001 - 6/2010	8/2001 - 6/2014
# commits	51500	173559	18099	20744	152296
# detected bug fixing commits	41227	53599	7744	8504	49388
# Type II bug fixing commits	20389 (49.5%)	19111 (35.7%)	3960 (51.1%)	4523 (53.2%)	10530 (21.3%)
# bug reports	27349	41633	5176	5374	43326
# Type I bug reports	20838 (76.2%)	34488 (82.8%)	3784 (73.1%)	3981 (74.1%)	38858 (89.7%)
# Type II bug reports	6511 (23.8%)	7145 (17.2%)	1392 (26.9%)	1393 (25.9%)	4468 (10.3%)
# re-opened bug reports	2876 (10.5%)	5681 (13.6%)	707 (13.7%)	653 (12.2%)	2311(5.3%)
Max # of fixing attempts for a bug report	97	56	24	45	36
Max # of fixing days for a bug report	1125	3781	1616	1947	889
Max # of involved developers for a bug report	6	7	14	12	6

reveals a limitation of the current identification algorithm for supplementary bug fixes.

On average, more than one tenth of bug fixes have been re-opened. Because our re-opened bugs are detected from both VCS and bug repositories, we can guarantee that any bug fix that has been re-opened can be identified. The proportion of re-opened bugs over all detected bug fixes are similar between projects, *i.e.*, from 5.3% to 13.7%.

Most bugs required only 1 to 2 fixing attempts and less than 24 hours to get fixed. Figure 3.2 shows the distribution of fixing attempts required for bugs. In the worst case, in Mozilla, a bug can require up to 97 attempts before getting fixed. In other projects, we also found bugs fixed with 24 to 56 attempts. To understand the period of time needed to make the supplementary fixes, Figure 3.3 presents the distribution of fix duration required for bugs. Overall, most bugs are solved within 24 hours (*i.e.*, 1 day). The maximum time taken for fixing bugs is 889 to 3,781 days. Some of those outliers (*e.g.*, bug #3875 in Netbeans) correspond to bugs where developers forgot to close a fixed bug report (this is a threat to validity), whereas others (*e.g.*, bug #55701 in Netbeans) really took such a long time to get fixed.

In Mozilla, Netbeans, Eclipse JDT Core, and Eclipse SWT, the proportion of Type II bugs is between 17.2% and 26.9%. This result is similar to the finding of Park et al. [3] in which Type II bugs account for 22.5% to 32.8% of all detected bugs. In Webkit, Type II bugs only account for 10.3% of all bugs. With a manual check, we found that Webkit allows developers to use both SVN and Git clients to access the source code. As a result, many commit messages mention an SVN style revision number instead of a Git revision number or a bug ID, making it difficult to track all commits related to a bug. For example, the following message could not be mapped to a bug report: “Rebaseline compositing/geometry/horizontal-scroll-

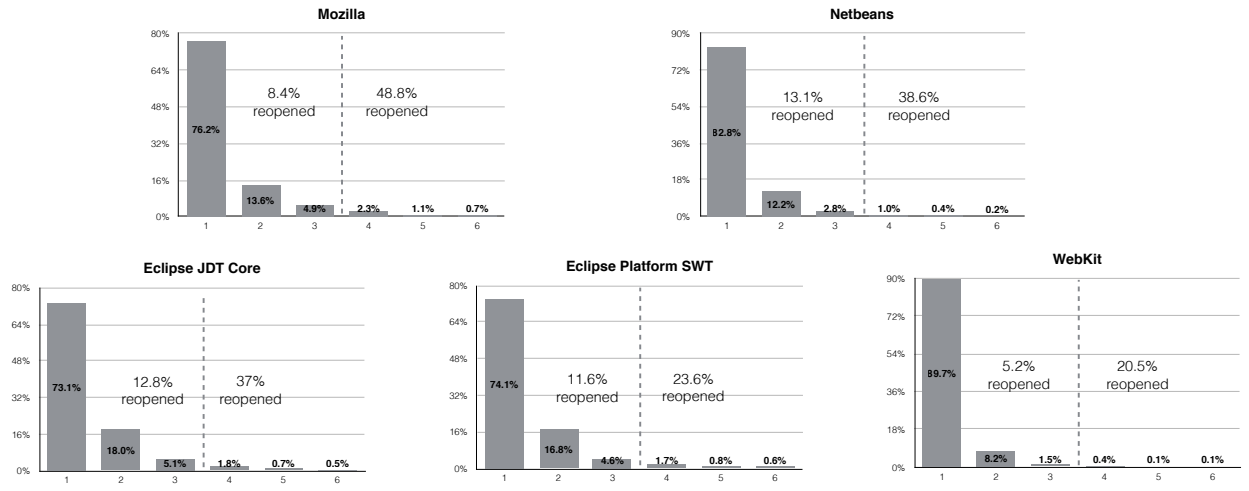

Figure 3.2 Number of fixes required for bugs as well as percentage of bugs that are re-opened within 3 fixing attempts and with more than 3 attempts

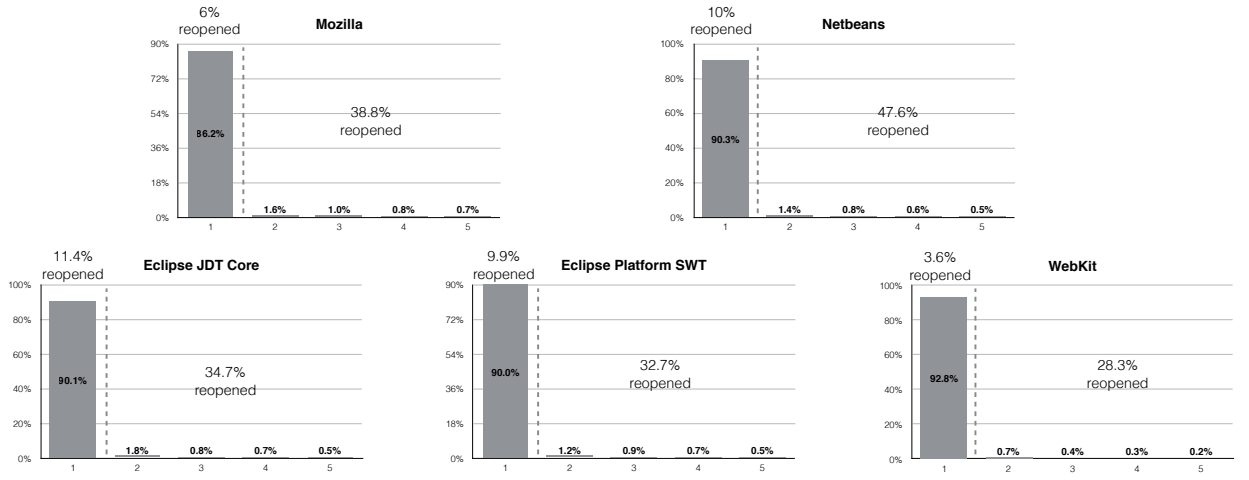

Figure 3.3 Number of fixing days of bugs as well as percentage of re-opened bugs that are fixed within 1 day and more than 1 day

composited.html after r107389". The latter number is an SVN style revision number.

Overall, in our five studied projects, supplementary bug fixes account for 10.3% to 26.9%, while re-opened bugs account for 5.3% to 13.7%.

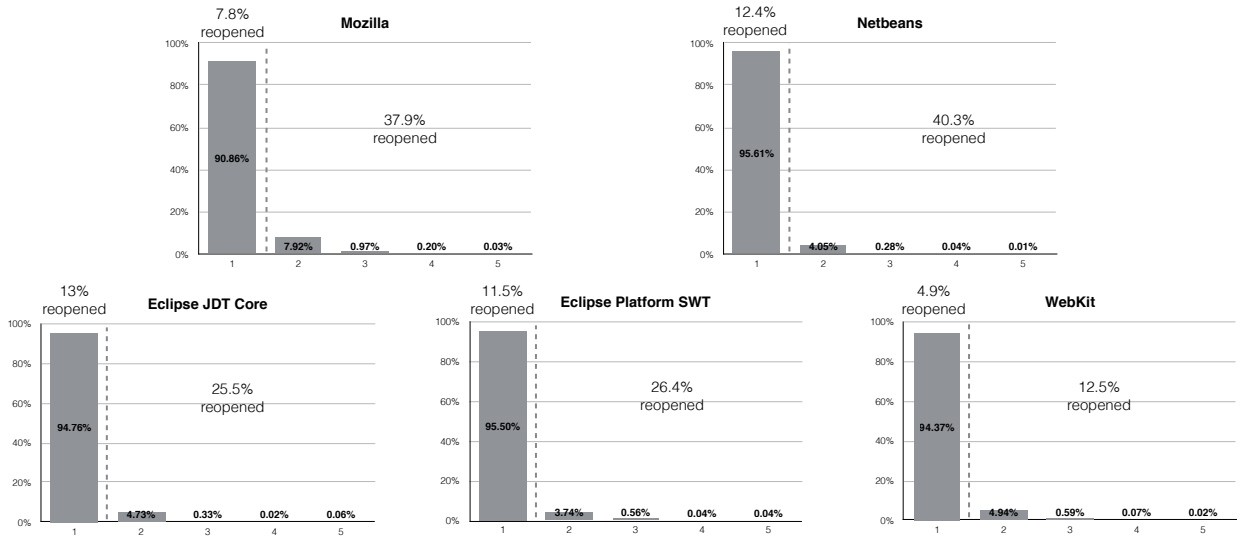

Figure 3.4 Number of developers participating in fixing bugs as well as percentage of re-opened bugs that are fixed by one developer and by multiple developers

RQ2: What is the relation between supplementary bug fixes and re-opened bugs?

Motivation. Many factors can explain the supplementary fixes found in our subject systems. A first explanation could be agile development and continuous integration practices that advocate for incremental development, in particular those that solved bugs within 24 hours, because developers may have just submitted their bug fixes incrementally (*i.e.*, through successive chunks of commits). A second explanation is suggested by the Type II bugs that experienced multiple bug fixing attempts over long period of time (up to 3,781 days). It is possible that long fixing period may increase the probability of bug re-opening. A third explanation is that multiple failing attempts at fixing a bug (many supplementary bug fixes) increases the odds that the bug will be re-opened in the future. A fourth explanation is that a bug fixing process may involve multiple committers. These multiple reasons can explain why different committers would contribute fixes for a same bug, such as the turnover in development teams or the complexity of a bug that may require the collaboration of several developers. To verify these hypotheses, this research question investigates the relation between Type II bug fixes and re-opened bugs.

Approach. To verify the above mentioned hypotheses regarding the relation between supplementary bug fixes and re-opened bugs, we split the results in Figure 3.2 and 3.3 in two parts (by dashed lines) to distinguish bugs that required less than three fixing attempts, and those that required more than three fixing attempts (respectively bugs fixed within 24

hours and those that required more than 24 hours). We choose these thresholds because they correspond to the modes of the distributions of the number of fixing attempts (respectively the number of fixing days). Also, bugs fixed by less than three successive commits, within 24 hours, are more likely to be linked to agile development rather than incorrect bug fixes. We then calculate the percentage of bugs below and over the above thresholds (on the left and right side of the dashed line), which are re-opened. The resulting percentages show where re-opened bugs are concentrated the most.

Figure 3.4 shows the distribution of the numbers of developers involved in fixing each bug. Dashed lines separate fixes by single developers and multiple developers. For each Type II bug, we count all different names or emails that appeared in the same Type II bug fix group (*i.e.*, all the fixing commits of a Type II bug) to identify the number of developers involved in fixing the bug. Because we extract this information from commit logs, it is possible that these developers (*i.e.*, the committers) are not the authors of the bug fixes but are instead reviewers with commit privileges [38]. A re-opened bug may also be assigned to a different (more experienced) developer in an attempt to avoid further fixing failures. To evaluate this last hypothesis, we investigate the distribution of re-opened bugs among the groups of bugs fixed by single versus multiple developers.

Figure 3.5 shows the relation between supplementary bug fixes and re-opened bugs. The green circles represent Type II bugs, blue circles represent re-opened bugs, pink circles represent "invalid reports" (*i.e.*, bug reports that have been closed by the following resolutions: "invalid", "wontfix", "duplicate", or "worksforme", which have a strong probability of being re-opened [2]). The overlapped parts are their intersection. For example, in Mozilla, there are 6,511 Type II bug reports (4,583 + 1,757 + 171), from which 1,928 are also re-opened bugs (1,757 + 171). Also, 171 of these re-opened bugs were "invalid reports" before being re-opened. We also found 948 re-opened bugs (324+624) with only one commit, among which 324 were "invalid reports" before being re-opened.

Findings. Re-opened bugs are more concentrated respectively in the areas above 24 hours, three fixing attempts, or by multiple developers. Overall, between 21.6% and 33.8% of Type II bugs have been re-opened at least once. However, almost half of the re-opened bugs were not detected as Type II bug fixes (*i.e.*, we did not find more than one fix for these re-opened bugs). This finding is quite a surprise because we expected re-opened bugs to be a subset of supplementary fixes. At first sight, this finding could be explained by limitations in our data set, such as developers forgetting to mention a bug ID in their commit message. However, closer analysis shows that 22.8% to 49.1% of re-opened bugs with only one fix tend to be linked with invalid reports, *i.e.*, not all re-opened bugs address previously fixed bugs. This

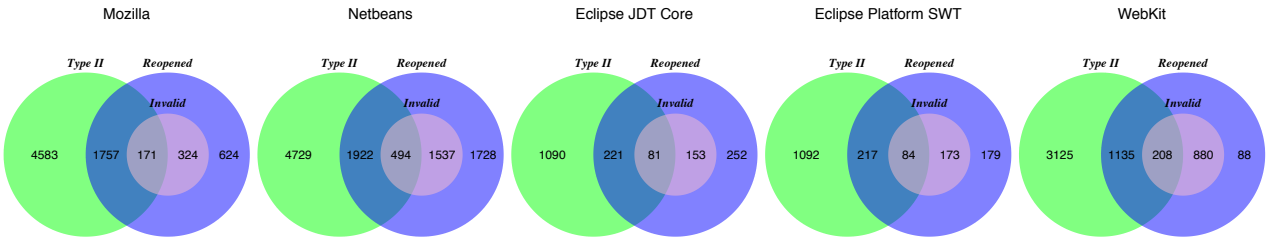

Figure 3.5 Relationship between supplementary bugs and re-opened bugs

seems counterintuitive, but in many cases the original bug fix was prematurely closed because developers considered that the problem described is not a bug (marked *invalid* in Bugzilla), the bug do not need to be fixed (marked as *wontfix*), the problem is a duplicate of an existing bug (marked as *duplicate*), or all attempts at reproducing this bug were unsuccessful (marked as *worksforme*). To validate whether the invalid reports are significantly associated with single re-opened bugs, we applied Chi-squared test and Fisher's exact test to compare the four types of invalid reports in re-opened bugs with only one commit and in those with multiple commits. The result shows that in all studied systems, the p-value is less than 0.05, *i.e.*, invalid reports have a significant association with single re-opened bugs. This finding also explains that not all bug re-openings have a negative impact on software development, contrary to the conclusion of earlier work [2]. In our subject systems, 22.8% to 49.1% of single re-opened bugs (*i.e.*, re-opened bugs with only one commit associated) have at least one of these invalid closed status. Those bugs have less impact on software quality than the re-opened bugs previously closed by the "fixed" status. Therefore, instead of building predictive models for bug re-opening over all bug fixes, like in [2], we only predict bug re-opening for supplementary bug fixes.

Between 21.6% and 33.8% of the studied Type II bugs have been re-opened, which tend to be fixed during long period, with multiple attempts or with multiple developers. Counterintuitively, almost half of the re-opened bugs have only one fixing attempt. 22.8% to 49.1% of these single re-opened bugs are due to prematurely closed reports.

RQ3: Can we predict the re-opening of supplementary bug fixes?

Motivation. In **RQ1** and **RQ2**, we observed that between 10.3% and 26.9% of bugs required at least one supplementary fix before they were resolved for good. Among these bugs that required supplementary fixes, between 21.6% and 33.8% were re-opened. Bug fix failures, and most of the re-opened bugs, are not desirable because they increase maintenance

Table 3.3 Work habit dimension

Attribute	**Explanation and Rationale**
Hour	Hour (0-24). Fix committed at certain hours may induce bug re-opening (*e.g.*, hours around quitting time).
Week day	Day of week (from Mon to Sun). Fix committed on certain week days may induce bug re-opening (*e.g.*, Friday) [39, 40].
Month day	Day in month (1-31). Fix committed on certain days may induce bug re-opening (*e.g.*, some dates before holidays).
Month	Month of year (1-12). Fix committed in some months may induce bug re-opening (*e.g.*, December, when we have Chrismas)
Day of year*	Day of year (1-366). Combining the rationales of month day and month.
Commit Size	Words in commit message. Too short (due to hasty work) or too long message (due to the difficulty) may lead to bug re-opening.

** this attribute was eliminated according to the VIF result*

Table 3.4 Bug report dimension

Attribute	**Explanation and Rationale**
Platform	Platform (*e.g.*, PC, Mac) on which the bug was reported. Bugs on some platforms are difficult to be solved, which may induce bug re-opening.
Severity	Severity of a bug report. Developers may mark a difficult bug as higher severity.
Priority	Priority of a bug report. Developers may mark a difficult bug as higher priority.
CC Number	Number of users who may not have a direct role to play on the bug, but who are interested in its progress. Bugs followed by many people may have a higher re-opening probability.
Description Size	Words in bug description. Too short (due to hasty work) or too long message (due to the difficulty) may lead to bug re-opening.
Invalid Status	Boolean value, *i.e.*, whether it exists an invalid status (see Section 3) before a commit. Invalid status may be followed by bug re-opening.

Table 3.5 Bug fix dimension

Attribute	**Explanation and Rationale**
Changed files	Number of changed files in a commit. Large number of changed files may increase the risk of bug re-opening.
Churn	Total number of inserted and deleted lines. Large number of changed LOC may increase the risk of bug re-opening.
Fixing time	Time span since the first fix. Long fixing time may induce bug re-opening.
Keywords	Some keywords (*e.g.*, crash, error, incorrect) in the commit message may imply bug re-opening.

Table 3.6 People dimension

Attribute	**Explanation and Rationale**
Reporter experience	The number of prior reported bugs. Inexperienced reporters are likely to introduce buggy report.
Assignee experience	The number of prior assigned bugs. Inexperienced assignee are likely to introduce buggy fixes.
Committer experience	The number of prior committed patches. Inexperienced committers are likely to introduce buggy fixes.

costs, degrade software quality and users' satisfaction [2]. For example, the average time from bug report to bug closing in one of the Eclipse projects for re-opened bugs was found to be as much as twice the average time to resolve a non-reopened bug [2]. In this research question, we replicate the work of Shihab et al. [2] to explore statistical models to predict whether or not a bug that required supplementary fixes will be re-opened. Using a prediction model, development teams will be able to target faulty/incomplete bug fixes for more thorough reviews, preventing re-opened bugs.

Approach. Based on the approach of Shihab et al. [2], we extract 19 attributes from commit logs and bug repositories along the four dimensions shown in Table 3.3 to Table 3.6.

We choose several regression and classification algorithms in R to build predictive models: General Linear Model (GLM), C5.0, ctree, cforest and randomForest. GLM is an extension of multiple linear regression for a single dependent variable. It is extensively used in regression analyses. The model C4.5 obtained a good prediction score in the work of Shihab et al. [2]. As a comparison, we use two Decision Tree models, C5.0 and ctree. C5.0 is an improved version of C4.5. The two algorithms are respectively derived from R packages "C50" and "party". In addition, we apply two implementations of the Random Forest algorithm, *i.e.*, randomForest from the R package "randomForest" and cforest from the R package "party". Random Forest was developed by Leo Breiman and Adele Cutler [36] and uses a majority voting of decision trees to generate classification (predicting, often binary, class labels) or regression (predicting numerical values) results. Random Forest offers good out-of-the-box performance and has performed very well in different defect prediction benchmarks [41]. The algorithm yields an ensemble that can achieve both low bias and low variance [42]. In our configuration, we build 50 trees with five randomly selected attributes in each tree.

Before building our models, we use Variance Inflation Factor (VIF) analysis to remove correlated variables. We set the correlation threshold to 5. Variables with VIF result over the threshold are considered as correlated, and hence are not included in our models. Among the selected attributes, "day of year" was eliminated, because its VIF result is higher than

5. We do not use reporter and assignee names like in Shihab et al.'s work [2], because these variables may lead to overfitted models.

To evaluate the importance of the different attributes (prediction variables), we applied the *MeanDecreaseGini* criteria, in which a higher value represents higher importance.

We applied 10-fold cross validation [43] to calculate the accuracy as well as the precision, recall, and F-measure for re-opened and non re-opened bugs. In the cross validation, each data set is randomly split into ten folds. Nine folds are used as the training set, and the remaining one fold is used as the testing set. We repeat the 10-fold cross validation for 10 times and report the average results obtained.

Findings. In all studied projects, randomForest outperforms C5.0 and other algorithms in accuracy, re-opened F-measure and non re-opened F-measure. When predicting re-opened bugs with randomForest, we can achieve an average accuracy of 84.0%, a precision of 83.6%, and a recall of 57.9%. Table 3.7 presents accuracy, precision, recall, and F-measure results for the five models predicting whether or not a bug that required supplementary fixes will be re-opened. Table 3.8 shows the top and second important attributes as well as their frequency in randomForest. As we executed 10 times the validation, the maximum frequency is 10. Overall, assignee experience, commit month, churn, reporter experience and committer experience are evaluated as the top or second attributes in different projects.

Our predictive models for re-opened bugs can achieve a precision between 72.2% and 97.0% and a recall between 47.7% and 65.3%. randomForest obtains the best prediction results.

3.3 Discussion

This section discusses some of the key aspects of our study that differ from the works of Park et al. and Shihab et al.

Identification of Supplementary Bug Fixes. During our data collection and processing, we have uncovered some limitations of the algorithm proposed by Park et al. [3] to track supplementary bug fixes. We have proposed an enhanced heuristic that can identify supplementary fixes with higher precision. Indeed, the new heuristic can track bug IDs that cannot be tracked by the algorithm proposed by Park et al. and it cross-checks all bug IDs mentioned in commit logs with the Bugzilla repositories to eliminate false bug IDs. Compared to the results of Park et al., the new heuristic have reported a higher percentage of supplementary bug fixes in Eclipse Platform SWT (25.9% vs. 24.0%) and Eclipse JDT core (26.9% vs. 22.5%), but a lower percentage in Mozilla (23.8% vs. 32.8%).

Prediction Models. In **RQ2**, we observed that almost half of the re-opened bugs are fixed

Table 3.7 Accuracy, precision, recall and F-measure (in %) obtained from GLM, C5.0, ctree, cforest and randomForest

System	Algo.	Acc.	Reop. Pre.	Reop. Rec.	Reop. F-m.	Non Reop. Pre.	Non Reop. Rec.	Non Reop. F-m.
Mozilla	GLM	64.2	69.6	6.6	12.0	64.0	**98.3**	77.5
	C5.0	74.3	70.0	53.9	60.9	76.0	86.4	80.8
	ctree	68.9	62.8	40.2	49.1	70.8	85.9	77.6
	cforest	76.4	79.4	49.3	60.8	75.5	92.4	83.1
	randomFor.	**82.1**	**82.8**	**65.3**	**73.1**	**81.8**	92.0	**86.6**
Netbeans	GLM	69.9	**87.7**	13.2	22.9	69.0	**99.1**	81.3
	C5.0	74.4	67.9	46.2	55.0	76.3	88.8	82.1
	ctree	71.0	75.0	21.8	33.8	70.6	96.3	81.5
	cforest	74.1	82.8	29.9	44.0	72.9	96.8	83.2
	randomFor.	**78.3**	80.2	**47.7**	**59.8**	**77.8**	94.0	**85.1**
JDT Core	GLM	77.5	76.6	15.1	25.3	77.5	98.4	86.7
	C5.0	83.3	72.7	53.8	61.8	85.7	93.2	89.3
	ctree	81.0	78.4	34.2	47.6	81.4	96.8	88.4
	cforest	83.3	**92.2**	36.8	52.6	82.3	**99.0**	89.9
	randomFor.	**87.7**	89.9	**57.7**	**70.2**	**87.3**	97.8	**92.2**
Plat. SWT	GLM	78.9	71.4	5.9	10.8	79.1	99.3	88.1
	C5.0	88.0	80.9	58.9	68.1	89.3	96.1	92.6
	ctree	82.2	73.0	29.3	41.8	83.1	97.0	89.5
	cforest	86.2	**97.0**	37.8	54.4	85.2	**99.7**	91.8
	randomFor.	**91.6**	95.4	**64.4**	**76.9**	**90.9**	99.1	**94.8**
WebKit	GLM	71.9	51.7	5.2	9.4	72.4	**98.1**	83.3
	C50	76.6	62.0	44.0	51.5	80.2	89.4	84.6
	ctree	74.9	58.4	38.5	46.4	78.7	89.2	83.6
	cforest	77.8	**72.2**	34.7	46.9	78.7	94.7	86.0
	randomFor.	**80.5**	69.8	**54.4**	**61.1**	**83.5**	90.8	**87.0**

Table 3.8 Top and second attributes and their frequency in randomForest

Project	Top attribute	Freq.	Second attribute	Freq.
Mozilla	commit month	5	commit month	5
	assignee exp.	5	assignee exp.	5
Netbeans	assignee exp.	10	reporter exp.	6
			commit month	3
			committer exp.	1
JDT	assignee exp.	10	commit month	10
SWT	assignee exp.	10	commit month	10
WebKit	churn	10	assignee exp.	7
			commit month	3

by only one commit. 22.8% to 49.1% of these single re-opened bugs were due to prematurely closed reports. These prematurely closed bugs do not necessarily have a negative impact on software development, because they are not related to failed bug fixes. For this reason, we decided in this study to focus our prediction of bug re-openings on supplementary bug fixes rather than on all bug reports as in the work of Shihab et al. [2]. Compared to their results (although our prediction models have a different dependent variable), our prediction models have a higher precision (72.2-97% vs. 52.1%-78.6%) and a lower recall (47.7%-65.3% vs. 70.5%-94.1%).

3.4 Threats to Validity

This section discusses the threats to validity of our study following the guidelines for case study research [44].

Construct validity threats concern the relation between theory and observation. We answered RQ1, RQ2, and RQ3 by carefully choosing the experimental measures, *i.e.*, identification technique and prediction algorithms. Concerning the proportion of supplementary fixes in a project, because our results for Webkit are different from those obtained by Park et al. [3], we have manually verified 200 commit messages of each project to validate the correctness of the results. Compared to Park et al., we enhanced the identification heuristic and cross-checked all the bug IDs obtained from commit logs with Bugzilla repositories to ensure that all detected bug IDs represent actual bug reports. In addition, we have the lowest Type II bug reports percentage in Netbeans, because in this project, many report messages are either non bug-fixing related or too brief, so it is difficult to map a fix to a certain bug ID. In WebKit, the re-opened bugs only account for 5.3% of all bug reports. In this project, many bug reports are only available to the internal staff (marked by "Access Denied"). Therefore, we cannot judge whether these bugs have been re-opened.

Internal validity threats concern factors that may affect a dependent variable and were not considered in the study. Theoretically, one would expect that all re-opened bugs are fixed more than once (*i.e.*, a fix before re-opening and other fixes afterwards), yet we obtained 33% to 57.5% re-opened bugs in the Type I bug set. Although we found that a large part of these bugs had been closed prematurely without any fix, another explanation could be a limitation of the identification technique by regular expressions. Even though we used bug IDs to trace a bug and revision numbers to map revisions and bug fixes. Some software organisations do not explicitly mark bug IDs in the revision history (or at least do not enforce this). So, we cannot track these bug fixes in VCS. In future work, we must explore novel identification heuristics. Another threat is related to the computation of bug fixing time values. It is possible that some developers forgot to close a fixed bug report.

Conclusion validity threats concern the relation between the treatment and the outcome. We paid attention not to violate assumptions of the constructed statistical models. According to the bug identification technique, we improved the existing heuristic, considered commits referring to an earlier commit's revision number, and compared the identified numbers with bug repositories. We manually checked the number sequences that were not detected by our mentioned regular expressions and found that none of those numbers were related to bug IDs. In the prediction, our best model, randomForest, can achieve a precision between

69.8% and 95.4%, a recall between 47.7% and 65.3%. Due to the state of the art of the bug identification technique from VCS, many bug fixes are not mapped to their corresponding bug reports. This may affect the recall of the prediction for bug re-opening.

External validity threats concern the possibility to generalise our results. Besides the project used by Park et al. [3] and Shihab et al. [2], we introduced two other projects in this study, *i.e.*, Netbeans and WebKit. They have a similar Type II bug percentage and a similar prediction accuracy. In future work, we plan to expand this study by analysing other open-source projects and applying novel identification techniques. For example, we could compare the bug fix committed time with the time in the attachments of bug reports to map a bug fix to its corresponding bug report. In addition, manual analysis of commit information and re-opening distribution will help us to determine the failure-prone fixes over all supplementary bug fixes. We provide our data and script in Github (`https://github.com/anlepoly/supplementary_fixes`). Researchers and software practitioners can verify our results or apply our approach to other projects.

3.5 Chapter Summary

In software development, bug fixing is a dominant activity for developers. A typical bug fixing cycle includes the reporting of the bug, the production of a fix, the verification of the fix, and the closing of the bug. However, sometimes, a closed bug later may be re-opened by developers. Previous studies show that such bug re-opening can increase the maintenance costs as well as degrade the software quality and the satisfaction of users. To discover the relation between supplementary bug fixes and re-opened bugs, we investigate supplementary bug fixes where more than one fix are associated with the same bug and re-opened bugs in five open-source projects and found that supplementary bug fixes account for 10.3% to 26.9% of total bug reports. In addition, in the subject systems, a high percentage (*i.e.*, from 21.6% to 33.8%) of the supplementary fixes have been re-opened. To help development teams target faulty/incomplete bug fixes (for more thorough reviews) and prevent re-opened bugs, we have explored the possibility of predicting bug re-openings over supplementary bug fixes, using GLM, C5.0, ctree, cforest and randomForest models. Results show that these models can achieve between 72.2% and 97.0% precision as well as between 47.7% and 65.3% recall. Moreover, we found between 33.0% to 57.5% of re-opened bugs with only one commit associated to them. These re-opened bugs have a strong association with invalid bug reports in all our five studied systems. In fact, they were prematurely dismissed as "invalid" before being re-opened. These bugs are not as risky as re-opened bugs with more than one commit to the software development. In other words, contrary to claims by existing works on re-

opened bugs, they will not affect the quality of the software product. Future researchers and practitioners who are mining data repositories can use our models to identify fault-prone bug fixes.

CHAPTER 4 HIGHLY-IMPACTFUL BUGS

Today, many software organisations (*e.g.*, Microsoft[1] and Mozilla[2]) embed automatic crash reporting tools in their software systems. Whenever the software crashes (*i.e.*, terminates unexpectedly in the user environment), the automatic crash reporting tool collects information about the crash and sends a detailed crash report to the software vendor. A crash report usually contains a signature, the stack trace of the failing thread, runtime information, such as the crash time, and information about the user environment, *e.g.*, the operating system, the version, and the install age. Crashes with the same signature are grouped together and filed in the same bug report. Software quality managers and developers usually judge the priority and severity of a bug by looking at the frequency of its related crashes [18].

Indeed, crash frequency is an important factor to evaluate the severity of a bug, because a high crashing frequency represents a high number of crash occurrences. However, frequency alone does not show the full picture because the crashes due to a bug may be concentrated on a limited user group, while the crashes due to another bug may affect most users of the software system. If the two bugs have the same number of crash occurrences, the second bug should be assigned a higher priority and severity because it impacts a larger user base.

Khomh et al. [4] have proposed an entropy metric to capture the distribution of crash occurrences among the users of a software system. They also proposed an entropy-based crash triaging approach that assigns a high priority to the bugs related to crashes that occur frequently (*i.e.*, high frequency) and affect a large user base (*i.e.*, high entropy).

In this work, we refer to crash-related bugs with both high crashing frequency and entropy as *highly-impactful bugs*. Although the entropy-based crash triaging approach proposed by Khomh et al. can successfully identify highly-impactful bugs, it takes a certain period of time to assess which crashes occur frequently with high entropy; a period during which the users of the system are negatively impacted by the crashes.

In this chapter, we investigate models to predict highly-impactful bugs early on before the software is released. We propose models that software organisations can use to identify highly-impactful bugs at an early stage of development, *e.g.*, during alpha or beta-testing phases. Such prediction models allow software organisations to focus on highly-impactful bugs earlier and improve the overall quality of their software systems effectively. We analyse the crash reports of Firefox and Fennec for Android (referred as Fennec in the rest of this

[1] http://www.microsoft.com/en-ca/default.aspx
[2] https://www.mozilla.org/en-US/

chapter) during the period of January 2012 to December 2012 and answer the following research questions:

RQ1: *What is the proportion of highly-impactful bugs?*

We apply the algorithm proposed by Khomh et al. [4] to identify users by their machines' configuration, *i.e.*, CPU type, OS name, and OS version, and found that highly-impactful bugs account for 42.3% of all bugs in Firefox and 37.9% in Fennec.

RQ2: *Do highly-impactful bugs possess different characteristics than other bugs?*

We study whether highly-impactful bugs possess different characteristics than other bugs (*i.e.*, bugs with low entropy and–or low frequency). Compared to other bugs, we observed that highly-impactful bugs are often fixed by more experienced developers and they generate larger amounts of comments. However, the proportion of highly-impactful bugs that are fixed and resolved is smaller in comparison to bugs with low entropy and–or low frequency.

RQ3: *Can we predict highly-impactful bugs?*

We applied GLM, C5.0, ctree, randomForest, and cforest algorithms to predict whether or not a bug will become highly-impactful, *i.e.*, it will have both a high crashing frequency and a high entropy. Our predictive models can achieve a precision up to 64.2% (in Firefox) and a recall up to 98.3% (in Fennec). Software organisations can use our prediction models in the early stage of their new releases to identify and fix bugs before they become highly-impactful.

RQ4: *Which benefits can be achieved by predicting highly-impactful bugs?*

The identification and correction of highly-impactful bugs at an early stage of development reduces the number of users impacted by these bugs; resulting in an improvement of the user-perceived quality of the software system. We calculate the date D_{pf} at which a highly-impactful bug, which is successfully predicted or transferred from a previous release, would be potentially fixed as follows. We compute the median fixing duration $Duration_{med}$ of fixed bugs that were assigned with the highest priority in the previous release and add it to the opening date (*i.e.*, D_o) of the highly-impactful bug, *i.e.*, $D_{pf} = D_o + Duration_{med}$. All the crashes that occurred after D_{pf} can be avoided if developers fix the bug without delay. Results show that some amounts of crash occurrences could be avoided with our prediction models. For Firefox and Fennec, on average, crash occurrences can be reduced by 23.0% and 13.4% respectively and the number of unique machine profiles that are impacted by crashes can be reduced by 28.6% and 49.4%, respectively.

Chapter Overview

Section 4.1 provides background information on Mozilla crash and bug triaging systems. Section 4.2 explains the identification of highly-impactful bugs. Section 4.3 presents our data collection and processing. Section 4.4 describes and discusses the results of the four research questions. Section 4.5 discusses threats to the validity. Section 4.6 summarises this chapter.

4.1 Mozilla Crash and Bug Triaging Systems

Mozilla ships software with a built-in automatic crash reporting tool, *i.e.*, the Mozilla Crash Reporter. When a Mozilla product crashes unexpectedly, the user receives a dialog box from the Mozilla Crash Reporting tool that suggests to submit the crash report to developers for improving the product's quality. A crash event and a detailed crash report are generated and sent to the Socorro server [19]. The crash report provides a stack trace for the failing thread and other information about the user's environment. A stack trace is an ordered set of frames where each frame refers to a method signature and provides a link to the corresponding source code. Figure 4.1 illustrates a sample crash report from Mozilla Firefox. Socorro collects crash reports from end users, assigns a unique ID to each report and groups similar crash reports together by the top method signatures of their stack trace. Such a group of crash reports in which all the stack traces share a common top frame is called a crash-type.

The Socorro server is an open-source project and its data are also open. It provides a rich Web interface for software practitioners to analyse crash-types. Developers can file bugs for crash-types with high crash occurrences in Bugzilla. Multiple crash-types can be linked to the same bug and multiple bugs can be linked to the same crash-type. In Soccoro, the list of bugs filed in Bugzilla is provided for each crash report. The Socorro server and Bugzilla are integrated, *i.e.*, developers can directly navigate to the linked bugs (in Bugzilla) from a crash-type summary in Socorro. Developers use the information contained in crash reports to debug and fix bugs. Figure 4.2 shows a general view of the Firefox crash triaging system.

Mozilla quality assurance teams triage bug reports and assign severity levels to the bugs [5]. Developers often port patches to fix a bug. Once approved, the patches are integrated into the source code.

4.2 Identification of Highly-impactful Bugs

We identify highly-impactful bugs following the approach proposed by Khomh et al. [4]. The approach is composed of three parts. The first part consists in identifying the list of unique user profiles that are impacted by each bug. The second part is the computation of the entropy and frequency of the bugs. The third part is the classification of bugs based on

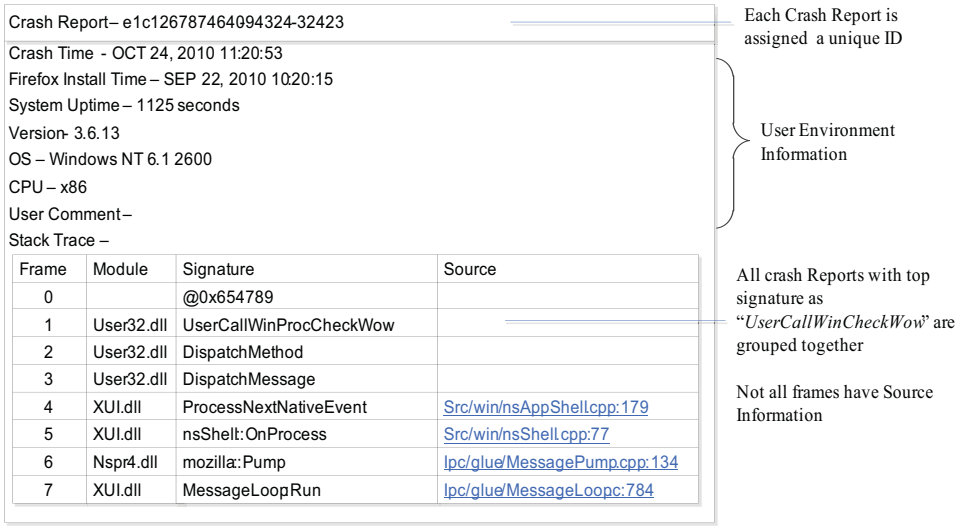

Figure 4.1 A sample crash report from Firefox

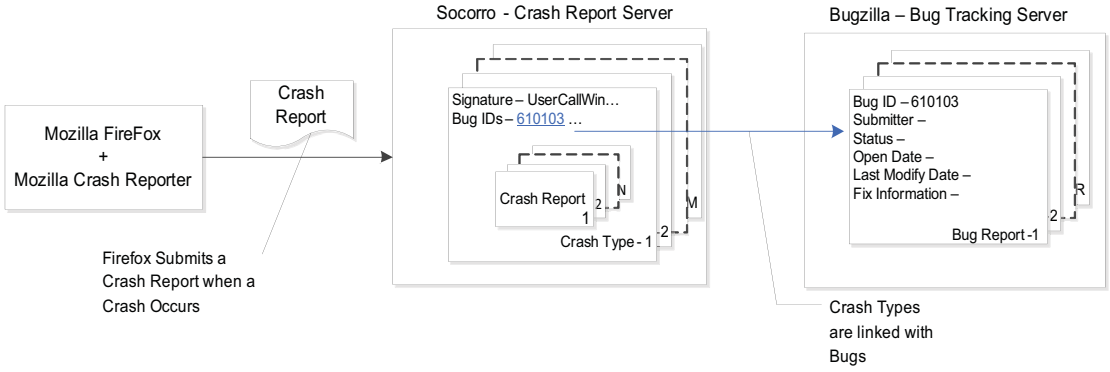

Figure 4.2 Mozilla crash triaging system

entropy and frequency values. The following subsections elaborate on each of these parts.

Part 1: Identification of Unique User Profiles Impacted by Each Bug

Mozilla crash reports do not contain personal information to identify users reporting the crashes for privacy reasons but we can identify user profiles with the following information from crash reports:

- *Crash signature*: top method signature of a crash. Crashes with the same crash signature are grouped into one crash-type in Socorro [4].
- *OS name*: name of the operating system on which the crash occurred.

- *OS version*: version of the operating system on which the crash occurred.
- *CPU type*: family model of the CPU of the machine on which the crash occurred.
- *Bug list*: list of bugs related to the crash.

For each crash report, we combine the contained users' environment information (*i.e.*, OS name, OS version, CPU type) to build a vector of unique profiles where each profile represents a unique machine configuration. Identifying unique machine configurations is important to compute the entropy of a bug. We associate each unique profile with the list of bugs for which crash reports contain information corresponding to the profile. Some crashes occurred before their corresponding bugs' opening, we perform an indirect mapping from crashes to bugs via crash-types: we map each bug to a set of crash-types (*i.e.*, a group of crashes with the same crash signature), then map each crash-type to a set of crash reports (each crash report stands for a user occurrence). Concretely, we build two dictionaries, $dict_{bug}$ and $dict_{crash}$. In $dict_{bug}$, the key is a bug ID and the value is a set of crash signatures. In $dict_{crash}$, the key is a crash signature and the value is a stack[3] of crash reports. By associating the two dictionaries, we can map a bug to a set of corresponding crash reports, even including those lacking bug information (*i.e.*, crashes reported before the creation of the bugs). Algorithm 1 and 2 show the pseudocode to link a bug to the machine profiles suffering its related crashes.

Part 2: Computation of the Entropy and Frequency of Bugs

We compute the entropy of a bug using the normalised Shannon's entropy [45] defined in Equation (5.1):

$$H_n(b) = -\sum_{i=1}^{n} p_i \times log_n(p_i) \tag{4.1}$$

where b is a bug; p_i is the probability of a specific machine profile i reporting the bug b ($p_i \geq 0$, and $\sum_{i=1}^{n} p_i = 1$); and n is the total number of unique machine profiles running the software. For a bug b, where all the machine profiles have the same probability of reporting b, the entropy is maximal (*i.e.*, 1). On the contrary, if a bug b is reported by only one machine profile i, the entropy of b is minimal (*i.e.*, 0). Bugs with high entropy values are reported by more unique machine profiles. Therefore, a high entropy value for a bug means that the bug is experienced by a high percentage of users.

[3]Every new crash report / crash occurrence will be appended at the end of the structure.

Algorithm 1: Map different crashed users to a crash-type, and map different crash-types to a bug

```
   Input: signature, user, buglist
 1 if signature in dict_crash then
 2 |   stack_user ← dict_crash[signature]
 3 |   add user to stack_user
 4 else
 5 |   stack_user ← new stack with user
 6 |   dict_crash[signature] ← stack_user
 7 foreach bug in buglist do
 8 |   if bug in bug_dict then
 9 |   |   set_signature ← dict_bug[bug]
10 |   |   add signature to set_signature
11 |   else
12 |   |   set_signature ← new Set with signature
13 |   |   dict_bug[bug] ← set_signature
```

Algorithm 2: Map crashed occurrences of machine profiles to their bugs

```
   Input: dict_bug
 1 foreach bug in dict_bug do
 2 |   set_signature ← dict_bug[bug]
 3 |   foreach signature in set_signature do
 4 |   |   stack_user ← dict_crash[signature]
 5 |   |   concatenate stack_user to occur_user
```

We compute the frequency of a bug b following Equation (4.2).

$$Fq(b) = \frac{Ncr(b)}{max_{j \in B}(Ncr(j))} \tag{4.2}$$

where b is a bug; $Ncr(j)$ is the number of crash reports linked to the bug j, and B is the total number of bugs. We implement the entropy and frequency computation algorithm in Python and share the code in the following repository: `https://github.com/swatlab/highly-impactful`.

Part 3 : Classification of Bugs Based on Entropy and Frequency Values

Similar to Khomh et al [4], we use the median values of entropy and frequency to classify bugs into the following four categories:

- **Highly-impactful Bugs:** bugs with frequency and entropy values above the median.

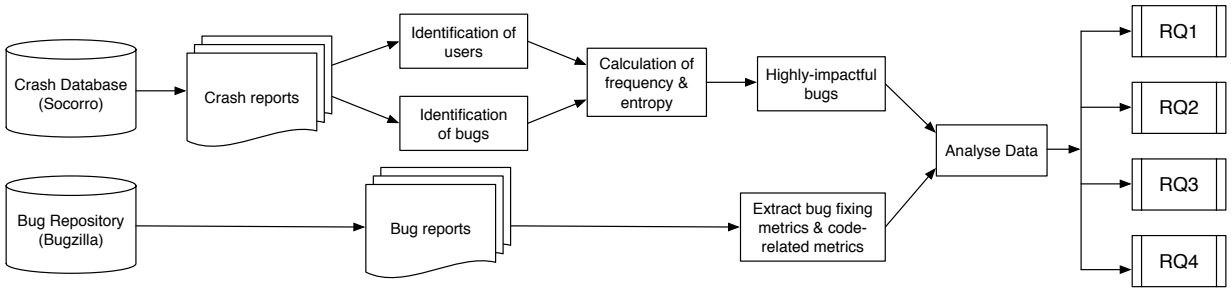

Figure 4.3 Overview of our approach to identify highly-impactful bugs and extract bug fixing metrics

These bugs impact a large proportion of users.

- **Skewed Bugs:** bugs with a high frequency value (*i.e.*, above the median) but a low entropy (*i.e.*, below or equal to the median). These bugs only seriously affect a small proportion of users and are more likely to be specific to the users' systems.
- **Moderately-impactful Bugs:** bugs that are highly-impactful among the users that report them (*i.e.*, entropy value above to the median) but do not occur very often to the majority of users (*i.e.*, frequency value below or equal to the median).
- **Isolated Bugs:** bugs with frequency and entropy values below or equal to the median. These bugs cause crashes rarely and affect a small number of users.

4.3 Study Design

This section presents the data collection and data processing of our case study, which aims to address the following four research questions:

1. What is the proportion of highly-impactful bugs?
2. Do highly-impactful bugs possess different characteristics than other bugs?
3. Can we predict highly-impactful bugs?
4. Which benefits can be achieved by predicting highly-impactful bugs?

4.3.1 Data Collection

We mine crash reports to compute the frequency and entropy of bugs as well as bug reports to study the characteristics of highly-impactful bugs and to build predictive models. We select the two following open-source software systems: *Firefox* and *Fennec for Android*. Firefox is a

popular Web browser developed by Mozilla Foundation. Fennec for Android is the codename of Firefox for Android. We analyse crash reports in both systems from January 2012 to December 2012 and the corresponding bug reports. These crashes and bugs are extracted from copies of Socorro and Bugzilla databases obtained from Mozilla Foundation. Table 4.1 shows the numbers of crash reports, extracted bugs reports, related releases, and detected users in the two subject systems.

Table 4.1 Numbers of crash reports, extracted bugs, related releases, and detected users in the studied systems

System	Crash reports	Bug reports	Releases*	Machine profiles
Firefox	132,484,824	6,636	22	40,942
Fennec	6,239,077	2,565	8†	11,488

* *We only count the number of official main releases, e.g., Firefox 10.0.1 and 10.0.2 are considered as minor releases of Firefox 10.*

† *Mozilla did not release Fennec 11, 12, and 13. The next release of Fennec 10.0 is Fennec 14.0. We consider all minor releases of Fennec 11, 12, and 13 as Fennec 10.*

4.3.2 Data Processing

Figure 4.3 shows a general view of our data processing approach. First, we mine crash reports to compute bug frequency and entropy values. Then, we mine bug reports to analyse the characteristics of highly-impactful bugs and build statistical models for their prediction. The remainder of this section elaborates more on these steps. All the data and scripts used in this chapter are available at: `https://github.com/swatlab/highly-impactful`.

Mining Crash reports

We parse crash reports using a Python script and extract the following information: *crash signature*, *OS name*, *OS version*, *CPU type*, *bug list*, and *uptime*. We use *uptime* as a predictor to answer RQ3, while using the rest of information to identify the list of unique machine profiles that are associated to each bug (see Section 4.2).

Mining Bug reports

Similar to crash reports, we parse bug reports using a Python script and extract information about bug opening date, bug fixing date, bug assigned date, reporter name, assignee name, involved fixer name(s), bug title, bug comments, and re-opened times. We also identify patches from bugs' attachments using the keyword "patch", *i.e.*, if the type of an attachment is marked as "patch", we consider it as a bug fixing patch.

Computing Code Complexity Metrics

We localise the faulty file(s) of each bug by analysing its crashing stack trace. In most crash-related bug reports, the top crashing frames are provided in the comments. We parse these crashing frames to extract crashed files or crashed method signatures, which are then mapped to the corresponding files in the source code of a specific release on which the crashes occurred. Using the SLOCCount tool [46], we found that, in both subject systems, C and C++ code accounts for more than 90%. Hence, in this study, we only take C and C++ files into account. We analyse every detected main source code release of Firefox and Fennec using the Understand tool [47], which generates its results in an Understand database (UDB) and provides a Python API for further analysis. We apply a Python script to extract five code complexity metrics for each faulty file identified: lines of code, average cyclomatic complexity, number of functions, maximum nesting, and ratio of comment lines over code lines. Detailed characteristics of these metrics are described in Section 4.4.

Social Networking Analysis Metrics

From the Understand databases generated in Section 4.3.2, we extract all C and C++ files and combine each .c or .cpp file with its corresponding .h file as a class node while merging their dependencies. To represent the relationship among these nodes, we build an adjacency matrix. Using this matrix and a Python script based on the network analysis package, igraph [48], we compute the following social networking analysis (SNA) metrics for each class node: PageRank, betweenness, closeness, indegree, and outdegree (detailed characteristics of these metrics are described in Section 4.4). We map the SNA metric values of each class node back to their corresponding bugs. Bugs that do not contain stack trace or could not be mapped to any file in the source code (*e.g.*, only crashed memory addresses are given in the stack trace) are not mapped to any SNA or complexity metric value.

4.4 Case Study Results

This section presents and discusses the results of our four research questions. For each question, we present the motivation, the approach followed to answer the questions, and the findings.

RQ1: What is the proportion of highly-impactful bugs?

Motivation. This question is preliminary to the other questions. It provides quantitative data on the proportion of highly-impactful bugs in our subject systems. This result will clarify the prevalence of highly-impactful bugs in Mozilla Firefox and Fennec to help understand the importance of their identification early on during the development process.

Approach. We identify highly-impactful bugs following the approach described in Section 4.2 and compute their percentage over the total number of reported bugs.

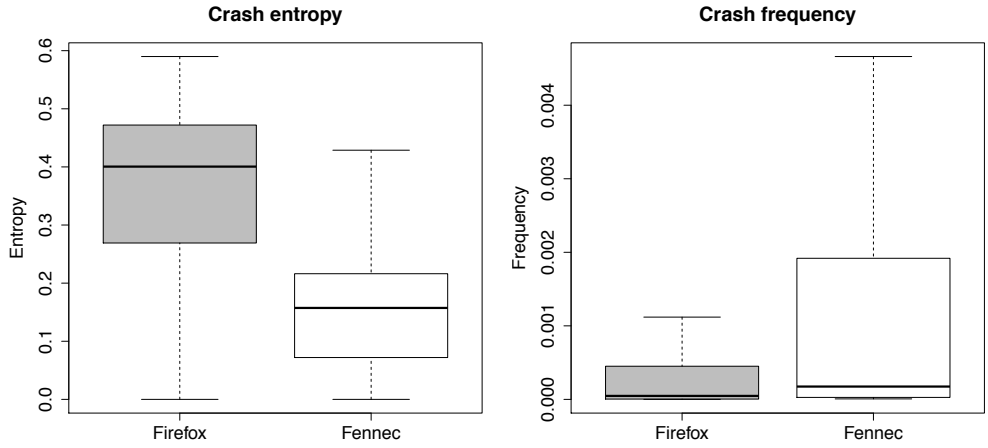

Figure 4.4 Distribution of bugs' crashing entropy and frequency in the subject systems

Table 4.2 Distribution of highly-impactful bugs, and other bugs in the subject systems

System	Highly	Skewed	Moderately	Isolated
Firefox	2806 (42.3%)	510 (7.7%)	512 (7.7%)	2808 (42.3%)
Fennec	972 (37.9%)	302 (11.8%)	301 (11.7%)	990 (38.6%)

Findings. Figure 4.4 shows the distribution of entropy and frequency values for all bugs in our subject systems. Table 4.2 shows the detailed distribution of bugs in the four categories presented in Section 5.

Highly-impactful bugs account for respectively 42.3% and 37.9% of all crash related bugs in the two studied systems.

By focusing their maintenance effort on highly-impactful bugs, software organisations will significantly improve the user perceived quality of their software systems, *i.e.*, it will reduce the proportion of crash occurrences in the field.

RQ2: Do highly-impactful bugs possess different characteristics than other bugs?

Motivation. Highly-impactful bugs crash very frequently in the field and impact a large proportion of users. Once such bugs are discovered, developers must fix them as soon as possible, to reduce their negative impact. Khomh et al. [4] recommended assigning the highest priority to these bugs. However, to effectively fix highly-impactful bugs quickly and avoid them being tossed several times, *i.e.*, passed around from one developer to another, it is important to assign them to the right developers. Previous work [9] found that tossing bugs results in longer bug fixing time. Highly-impactful bugs also must be fixed completely, *i.e.*, without any re-opening, because re-opened bugs negatively impact software quality [2]. In this research question, we study the bug fix characteristics of highly-impactful bugs and compare them to non highly-impactful bugs (*i.e.*, all other remaining bugs).

Approach. To answer this research question, we first apply the approach described in Section 4.2 to classify bugs from our subject systems in four categories: Highly-impactful Bugs (highly), Skewed Bugs (skewed), Moderately-impactful Bugs (moderately), and Isolated Bugs (isolated). Next, for each bug, we parse the corresponding bug report using a Python script available at `https://github.com/swatlab/highly-impactful`, and compute the metrics shown in Table 4.3. Finally, we test the following 10 null hypotheses to compare the bug fix characteristics of highly-impactful bugs and bugs from the other three categories.

Comparing the effort required to fix highly-impactful bugs vs. other bugs.

H_{01}^{1}: *the fixing time is the same for highly-impactful bugs and other bugs.*

H_{01}^{2}: *the triaging time is the same for highly-impactful bugs and other bugs.*

H_{01}^{3}: *the number of patches is the same for highly-impactful bugs and other bugs.*

H_{01}^{4}: *the comment size is the same for highly-impactful bugs and other bugs.*

H_{01}^{5}: *the re-opened frequency is the same for highly-impactful bugs and other bugs.*

Comparing people involved in filing and fixing highly-impactful bugs vs. other bugs

H_{02}^{1}: *reporters' experience is the same for highly-impactful bugs and other bugs.*

H_{02}^{2}: *assignees' experience is the same for highly-impactful bugs and other bugs.*

H_{02}^{3}: *the number of fixers is the same for highly-impactful bugs and other bugs.*

H_{02}^{4}: *the number of developers interested in highly-impactful bugs is the same as other bugs.*

Comparing the bug fix rate of highly-impactful bugs vs. other bugs

H_{03}^{1}: *the percentage of highly-impactful bugs that are closed is the same as other bugs.*

Table 4.3 Metrics used to compare the characteristics of highly-impactful bugs and other bugs

Metric	**Description and Rationale**
Fixing time	Duration (in seconds) of the period between the bug opening date and the last modification date. We use the fixing time as a proxy for fixing effort.
Triaging time	Duration (in seconds) of the period between the bug opening date and the first bug assignment date. Low triaging time may imply an efficient work on bug classification.
Patch number	Number of patches submitted to fix a bug. A high number of patches means a high fixing effort (multiple attempts were made to fix the bug).
Comment size	Number of words in the comments contained in a bug report. A high number of words would mean that an intensive discussion took place.
Reporter experience	The number of bugs filed by the reporter of a bug in the past. A reporter who filed a high number of bugs is likely to gain recognition for the relevance of her bug reports. Quality managers may decide to pay more or less attention to her reported bugs.
Assignee experience	The number of bugs fixed by the bug assignee in the past. Assignees with a high experience are likely to fix the bug quickly.
Fixer number	The number of unique developer names in the bug fixing history. A high number of fixers means that the bug was tossed around or required the participation of multiple developers.
CC number	Number of developers who were interested in the bug. These developers may not have played a direct role in fixing the bug. However, a high CC number indicates a high interest in a bug.
Re-opened frequency	Number of time that a bug is re-opened. Frequent bug re-openings increase development costs and decrease users' satisfaction [2].
Closed percentage	Percentage of bugs from a category (*i.e.*, highly, skewed, moderately, or isolated) whose final status is "resolved" or "verified". A high closed percentage for a category may suggest a prioritisation of the bugs from the category.

We apply the Wilcoxon rank sum test [49] to accept or reject the first 9 hypotheses. For H_{03}^{1}, we compare the values of *closed percentage* obtained for the four categories. The Wilcoxon rank sum test is a non-parametric statistical test used for assessing whether two independent distributions have equally large values. We use this test to compare the characteristics of highly-impactful bugs with other bugs. We also apply the Kruskal-Wallis test [49] to compare the characteristics of bugs from all the four categories (*i.e.*, highly, skewed, moderately, and isolated). The Kruskal-Wallis test is an extension of the Wilcoxon rank sum test. It is used to assess whether two or more samples originate from the same distribution. It does not assume a normal distribution since it is a non-parametric statistical test. For all statistical tests, we use a 95% confidence level (*i.e.*, p-value < 0.05) to decide whether to reject a null

hypothesis. As we are investigating 9 characteristics, we apply the Bonferroni correction [50] which consists in dividing the threshold p-value by the number of tests (*i.e.*, we consider that there is a statistically significant difference only if the $p\text{-}value < 0.05/9 = 0.0056$).
Because highly-impactful bugs are defined using median (*i.e.*, 50th percentile) values of entropy and frequency (see Section 5), we perform a sensitivity analysis to assess the impact of this chosen threshold on the results. Precisely, we repeat the identification of highly-impactful bugs using the 70th, and 90th percentiles, and repeat testing the 10 null hypotheses mentioned above.

Findings. Table 4.4 shows the mean values of metrics described in Table 4.3, for highly-impactful bugs and bugs from the other three categories (*i.e.*, skewed, moderately, and isolated), as well as the p-values for Wilcoxon and Kruskal-Wallis tests. Statistically significant p-values are bolded in Table 4.3. On the one hand, the results of the Wilcoxon rank sum test are statistically significant for comment size and reporter experience in both studied systems. Therefore we reject H^4_{01} and H^1_{02}. Although in general highly-impactful bugs have longer comments and are reported by more experienced developers, in Firefox, their comment size and developers' experience are lower than those of skewed bugs. On the other hand, there is no statistically significant difference between highly-impactful bugs and other bugs for the *fixing time*, *triaging time* and the *fixer number* in both systems, hence we cannot reject H^1_{01}, H^2_{01} and H^3_{02}. For H^3_{01}, H^5_{01}, H^2_{02}, and H^4_{02} the results are system dependant. We discuss them in detail in the following paragraphs.

Statistically significant differences: In Firefox, the number of patches proposed for highly-impactful bugs is significantly lower than other bugs. The experience of developers assigned to highly-impactful bugs and the number of developers indirectly involved (*i.e.*, CC number) in fixing highly-impactful bugs are significantly higher than other bugs. In Fennec, the bug re-opening frequency of highly-impactful bugs is significantly higher than other bugs. In both systems, the proportion of highly-impactful bugs that are closed is slightly higher than the proportion of bugs from the skewed category, but significantly lower than the proportion of bugs from moderately-impactful and isolated categories that are closed. This result suggests that Mozilla developers do not necessarily prioritise highly-impactful bugs during bug fixing activities. Our finding is consistent with previous result by Kim et al. [18], which suggested that Mozilla developers judge the priority and severity of a bug mostly by looking at the frequency of its related crashes. The sensitivity analysis confirms these findings for highly-impactful bugs detected using the 70^{th} percentile. However, if we identify highly-impactful bugs using the 90^{th} percentile, the differences between the reporter experience and assignee experience of highly-impactful bugs and other bugs is not statistically significant in Firefox. In Fennec, the difference between the reporter experience of highly-impactful bugs

Table 4.4 Mean value of characteristic metrics for highly-impactful bugs and other bugs, as well as the p-values of the Wilcoxon and Kruskal-Wallis tests

System	Metric	Highly	Skewed	Moderately	Isolated	Wilcoxon	Kruskal-W.
Firefox	Fixing time	1.22e+7	1.40e+7	1.52e+7	1.55e+7	0.126	0.031
	Triaging time	4.51e+6	5.54e+6	7.37e+6	3.89e+6	0.212	0.028
	Patch number	0.48	0.52	0.61	0.68	**4.3e-14**	**<2.2e-16**
	Comment size	595.0	728.3	451.9	421.2	**<2.2e-16**	**<2.2e-16**
	Reopened freq.	0.063	0.061	0.065	0.069	0.645	0.858
	Reporter exp.	202.0	224.5	123.8	109.8	**<2.2e-16**	**<2.2e-16**
	Assignee exp.	959.8	1238.4	747.6	776.9	**5.0e-15**	**<2.2e-16**
	Fixer number	7.0	6.7	6.8	6.6	0.709	0.588
	CC number	6.7	6.5	6.1	5.7	**1.6e-5**	**1.0e-5**
	Closed %	59.2%	55.1%	73.5%	73.9%	–	–
Fennec	Fixing time	8.87e+6	6.63e+6	8.34e+6	1.33e+7	0.41	0.042
	Triaging time	3.12e+6	5.86e+6	1.36e+6	6.63e+6	0.197	0.34
	Patch number	0.59	0.43	0.80	0.68	0.066	**2.0e-6**
	Comment size	585.7	538.2	484.3	428.6	**7.4e-11**	**9.0e-11**
	Reopened freq.	0.122	0.08	0.070	0.066	**9.7e-6**	**1.4e-4**
	Reporter exp.	152.3	149.1	97.8	97.0	**1.7e-8**	**2.3e-13**
	Assignee exp.	341.7	459.4	276	284.1	0.036	**4.8e-9**
	Fixer number	6.8	5.8	7.0	6.8	0.112	**8.0e-4**
	CC number	6.5	6.0	6.4	6.2	0.174	0.25
	Closed %	63.7%	53.3%	73.2%	74.8%	–	–

and other bugs is not statistically significant. We attribute this result to the low number of highly-impactful bugs found in these systems when using the 90$^{\text{th}}$ percentile (*i.e.*, 4.2% in Firefox and 3% in Fennec).

Non statistically significant differences: The results from Table 4.4 show lower fixing time values in Firefox for highly-impactful bugs in comparison to other bugs. However, the Wilcoxon test was not statistically significant. In our previous work [4], we found that highly-impactful crash-types required statistically longer fixing time. However, a crash-type is often related to more than one bug (and vice-versa), which could explain the different result obtained here. Also, we relied on machine profiles to identify highly-impactful bugs instead of user profiles as in our previous work [4]. This choice was dictated by the data provided to us by the Mozilla Foundation, which did not contained references to users (because of privacy restrictions). The result of the sensitivity analysis shows that the number of developers involved in the correction of bugs (*i.e.*, the fixer number) with frequency and entropy values above the 70$^{\text{th}}$ percentile is statistically significantly higher than other bugs in Fennec. If we identify highly-impactful bugs using the 90$^{\text{th}}$ percentile, their fixing time is significantly higher than the fixing time of other bugs in Firefox. However, they are in small number (*i.e.*, 4.2% of total crash-related bugs).

All these results suggest that Mozilla quality assurance teams do not prioritise highly-impactful bugs (a lower proportion of these bugs are fixed) albeit they impact a large user base and do not seem to be more difficult to fix than other bugs. If developers could predict these highly-impactful bugs early on and fix them, they would significantly reduce their negative impact and improve the user-perceived quality of the system.

RQ3: Can we predict highly-impactful bugs?

Motivation. Khomh et al. [4] have proposed an entropy-based approach (described in Section 4.2) that can be used to identify highly-impactful bugs. However, this approach requires a certain period of time to assess which bug occurs frequently with high entropy. Table 4.4 shows that the average triaging time for highly-impactful bugs in Firefox and Fennec is respectively 52.2 days and 36.2 days. During those periods, the users of the systems are impacted by crashes that can lead to data loss and–or frustration. In this research question, we investigate strategies to predict highly-impactful bugs. Specifically, our goal is to determine whether a bug is highly-impactful at the moment it is reported (*i.e.*, once the bug report is filed). Such a prediction can be applied by software organisations to identify highly-impactful bugs at an early stage of development, *e.g.*, during alpha or beta-testing phases. This approach may allow developers to focus on highly-impactful bugs earlier and improve the overall quality of the software more efficiently.

Approach. We mine bug reports and crash reports and compute the metrics described in Table 4.5 to Table 4.8. We select these metrics because they have been successfully used in bug prediction studies (*e.g.*, Shihab et al. [2] used *Week day*, *Month day*, and *Day of year* to predict re-opened bugs) and they are available once a bug is submitted. As in Chapter 3, we choose several regression and classification algorithms in R to build predictive models: General Linear Model (GLM), C5.0, ctree, randomForest, and cforest.

Before building our models, we use the Variance Inflation Factor (VIF) analysis to remove correlated variables. We set the threshold to 5. Variables with VIF results over the threshold are considered correlated. In Table 4.5 to Table 4.8, * indicates for the eliminated metrics in Firefox while † indicates for the eliminated metrics in Fennec.

We cluster the extracted bugs of the different releases; grouping bugs from minor releases into their main release (*e.g.*, bugs from releases 10.0.1 and 10.0.2 are grouped with those of the major release 10). We intended to use consecutive releases to test the performance of our prediction models but observed that a high proportion of bugs is transferred across the releases. In some releases, the "transferred bugs" account for more than 80% of all bugs. This high bug transfer rate is due to Mozilla rapid release cycle, which it follows since 2011 [51].

Table 4.5 Bug report metrics

Metric	**Description and Rationale**
Week day	Day of week (from Mon to Sun). Bug reports created on certain week days may be overlooked for fixing; resulting into large numbers of crashes. (*e.g.*, Friday) [39, 40].
Month day	Day in month (1-31). Bug reports created on certain days may be overlooked for fixing; resulting into large numbers of crashes. (*e.g.*, some dates before holidays).
Month	Month of year (1-12). Bug reports created in some seasons may be overlooked for fixing; resulting into large numbers of crashes. (*e.g.*, December, during Chrismas holidays)
Day of year*†	Day of year (1-366). Combined the rationales of month day and month.
Description Size	Number of words in a bug description. A too short message (due to hasty work) or too long message (due to the difficulty) may lead to fixing failure and late resolution of the bug, which may lead to large numbers of crashes.
Component	Component where a bug is located. Bugs occurring in complex or central (*i.e.*, highly coupled with other parts of the system) components may be difficult to resolved, which may lead to large numbers of crashes.
Reporter experience	Number of bugs filed by the reporter of a bug in the past. A reporter who filed a high number of bugs is likely to gain recognition for the relevance of her bug reports. Quality managers may decide to pay more or less attention to her reported bugs, which may result into large or low numbers of crashes.

Table 4.6 Crash report metrics

Metric	**Description and Rationale**
Uptime	Median uptime of crashes related to a bug. The uptime of a crash is the duration (in seconds) of the period during which Firefox or Fennec was running on a user's machine before the occurrence of the crash. Bugs related to low uptime values may cause large numbers of crashes (a user may restart its system and–or the software multiple times in hope that a rejuvenation will suppress the crash).
Pre-opening daily crashes	Average daily crash occurrences for a bug before the bug report is filed (data extracted from crash reports during the period of February 2010 to December 2011). High pre-opening crash occurrences may imply high post-opening crash occurrences.
Pre-opening daily impacted users†	Average daily number of machine profiles impacted by a bug before the bug report is filed (calculated from the same dataset as pre-opening daily crashes). High pre-opening daily rate of impacted machine profiles is likely to translate into high post-opening rate of impacted machine profiles.

Table 4.7 Code complexity metrics

Metric	**Description and Rationale**
LOC	Lines of code of the bug-related class. A large class may be hard to maintain and prone to crashes.
# of functions[*†]	Number of functions in the bug-related class. Same rationale as LOC.
Cyclomatic complexity	Average cyclomatic complexity of the functions in the bug-related class. Complex code is hard to maintain and prone to crashes.
Max nesting[†]	Maximum level of nested functions. A high level of nesting increases the conditional complexity and is likely to increase the crashing probability.
Comment ratio	Ratio of the number of comments to the total lines of code in the bug-related class. A code with few comments may not be easy to understand, and may consequently lead to large numbers of crashes.

Table 4.8 Code complexity metrics (other selected metrics share the rationale as PageRank)

Metric	**Description and Rationale**
PageRank[*†]	Time fraction spent to "visit" the bug-related class in a random walk in the call graph. If an SNA metric of a class is high, a bug in that class may be triggered through multiple paths and the bug is likely to appear frequently, because multiple paths lead to that class.
Betweenness	In the call graph, number of classes passing through the bug-related class among all shortest paths.
Closeness	Sum of lengths of the shortest call paths between the bug-related class and all other classes.
Indegree	Numbers of callers of the bug-related class.
Outdegree	Numbers of callees of the bug-related class.

With short release cycles, many bugs are transferred from an old release to new releases before getting fixed. If only few new bugs are discovered in a new release there is no need for a prediction model because developers could manually triage the bugs. Hence, to test the performance of our proposed models, we consider releases with at least 25% of new bugs. More specifically we test our models on releases of Firefox and Fennec containing respectively 25%, 30%, and 35% of new bugs (in fact, there are only two releases of Firefox in which new bugs account for more than 40%). For each of these thresholds, we select Firefox and Fennec releases with amounts of new bugs greater than the threshold. We use the new bugs to create a testing set. We use the bugs from the preceding release to train the models. We measure the accuracy, precision, recall and F-measure for each classification algorithms using only the new bugs from the testing set (to avoid overfitting). We use the variable importance function (*e.g.*, varimp) in R to discover the top predictors for each of the algorithms.

Findings. Table 4.9 shows the average prediction accuracy, precision, recall, and F-measure

Table 4.9 Accuracy, precision, recall and F-measure (in %) obtained from GLM, C5.0, ctree, randomForest, and cforest to predict highly-impactful bugs (the proportion of new bugs is > 35% (testing set))

System	Metric	GLM	C5.0	ctree	randomForest	cforest
Firefox	Accuracy	40.2	64.2	**66.2**	64.9	63.1
	precision	31.7	60.3	**64.2**	61.7	58.8
	recall	23.2	70.6	64.0	66.9	**71.8**
	F-measure	26.8	**65.0**	64.1	64.2	64.7
Fennec	Accuracy	**52.2**	45.2	46.2	47.3	45.4
	precision	32.6	43.8	45.1	**45.5**	44.9
	recall	6.6	80.1	94.9	91.7	**98.3**
	F-measure	10.9	56.6	61.2	60.9	**61.6**

Table 4.10 Number of training/testing pairs, precision and recall (in %) of cforest with different proportions of new bugs (testing sets)

New bugs	Firefox			Fennec		
	# pairs	Precision	Recall	# pairs	Precision	Recall
25%	66	49.4	84.5	21	42.3	97.2
30%	34	49.2	83.1	17	43.2	97.4
35%	12	65.6	71.8	12	44.9	98.3

for the five classification algorithms in Firefox and Fennec.

cforest achieves the best recall when predicting highly-impactful bugs in both studied systems. In general, our predictive models can obtain a precision of 64.2% in Firefox and 45.4% in Fennec, as well as a recall of 71.8% in Firefox and 98.3% in Fennec.

Table 4.10 shows how the precision, recall, and F-measure of the cforest algorithm varies with the size of the testing set (*i.e.*, the amount of new bugs). Precision and recall increase with the size of the testing set in Fennec; in Firefox, the precision increases while the recall decreases.

The best results are obtained when the proportion of new bugs represents more than 35% of all bugs. We computed the top predictors for the different models and found that ***pre-opening daily impacted user number*** **is the most important predictor for four algorithms in Firefox; meaning that the pre-opening (*i.e.*, before the bug report is opened) impact of a bug on users is a good indicator of its future impact on users (*i.e.*, after bug opening)**, which is an expected result. In Fennec, *component of bug*, *bug opened month*, and *median crashing uptime* are the top predictors for at lease two algorithms. The best predictor is not obviously identified for this system. One explanation may be the small size of Fennec.

RQ4: Which benefits can be achieved by predicting highly-impactful bugs?

Motivation. Results from **RQ3** show that highly-impactful bugs can be predicted early on before a new release, with a recall of 71.8% in Mozilla and 98.3% in Fennec. Therefore, rather than waiting for a large number of crashes to occur, developers can identify and address highly-impactful bugs without delay. To quantify the benefits that can be achieved by predicting highly-impactful bugs, we simulate the application of our proposed cforest model (the best predictive model of the case study presented in RQ3) to the 12 pairs of releases that contain at least 35% of new bugs and assess the amount of crash occurrences and unique machine profiles that can be avoided.

Approach. In the studied training releases, we compute the median fixing time (*i.e.*, the period between the bug opening date and the last modification date) of all resolved bugs with the priority "P1". Those bugs are assigned the highest priority and are expected to be fixed earlier than other bugs. We refer to it as $duration_{med}$. For the bugs in the studied testing releases, we apply our cforest model to predict their categories (*i.e.*, highly, skewed, moderately, or isolated). We use Equation (4.3) to compute the simulated fixed date of a bug:

$$D_{pf} = D_o + Duration_{med} \tag{4.3}$$

where D_{pf} stands for the date at which a highly-impactful bug, which is successfully predicted or transferred from a training release, would be potentially fixed; D_o stands for the opened date of the bug; and $Duration_{med}$ stands for the median fixing duration of fixed bugs that were assigned with the highest priority (*i.e.*, P1). We consider all the crashes (related to the predicted or transferred highly-impactful bug) that occurred after the simulated fixing date, as crashes that can be avoided, if developers fix the bug without delay. We identify machine profiles impacted by these crashes by applying the heuristic described in Section 4.2.

To calculate the proportion of crash occurrences that can be reduced, we sum the crashes that can be avoided for all successfully predicted and transferred highly-impactful bugs in each testing release, then divide this number by the total number of crashes in the release. To calculate the proportion of unique machine profiles that can be reduced, for every testing release, we subtract each successfully predicted or transferred bug's related machine profiles that are impacted before the simulated fixing date from the total unique machine profiles impacted by this bug. We divide this number by the total number of machine profiles to obtain the percentage of reduced machine profiles for the bug. Next, we compute the average percentage of reduced machine profiles for all bugs in the testing release. Finally, we compute the average percentage of crash occurrences and unique machine profiles that can be reduced for Firefox and Fennec releases respectively.

Because developers' time and resource is limited, we also compute the amount of time that developers would spend fixing false positives (*i.e.*, wrongly predicted highly-impactful bugs). We divide the result by the total time spent on bug fixing activities to calculate the percentage of time lost to false positives.

Finding. **Some amounts of crash occurrences can be avoided by our "early triaging technique". On average, the numbers of crash reports can be reduced by 23.0% in Firefox and by 13.4% in Fennec. The numbers of unique machine profiles that are impacted by crashes can be reduced by 28.6% in Firefox and by 49.4% in Fennec.** In addition, false positive highly-impactful bugs would consume on average 6.3% of the total bug fixing time in Firefox (respectively 29.6% in Fennec). We manually investigated these false positive highly-impactful bugs and found that 96.4% of these bugs in Firefox (respectively 95.5% in Fennec) are assigned a severity level of "blocker" or "critical". Also, 51.2% of them eventually get fixed in Firefox (respectively 41.8% in Fennec). This result suggests that even though these false positives are not highly-impactful bugs in the sense that they do not have both high entropy and high frequency, they are nonetheless important and should be fixed in priority. Therefore, the amount of time spent on these bugs is not wasted.

In conclusion, these results show that triaging and predicting highly-impactful bugs early (before a new release of a software system) can help reduce a large amount of crashes experienced by users, which could improve the overall quality of the software system in a more cost-effective manner.

4.5 Threats to Validity

This section discusses the threats to validity of our study following the guidelines for case study research [44].

Construct validity threats concern the relation between theory and observation. In this study, the construct validity threats are mainly due to measurement errors. We parse crash and bug reports from copies of Socorro and Bugzilla databases obtained from Mozilla Foundation. Khomh et al. [4] found in a previous study that highly-impactful crash-types require longer fixing time. However, our result in this study show that the fixing time of highly-impactful bugs in Firefox is slightly lower in comparison to other bugs. We attribute this difference to the fact that a crash-type is often related to more than one bug (and vice-versa). Also, in this study, we rely on machine profiles to identify highly-impactful bugs instead of users' profiles inferred from installation time as in Khomh et al.'s work [4]. This choice is dictated by the data provided to us by Mozilla Foundation, which do not contain references to users.

In **RQ4**, we estimated the date at which a successfully predicted or transferred highly-impactful bug is fixed by adding the median fixing duration of fixed bugs that are assigned the highest priority in our training data set to the bug opening date. This estimation may not be accurate. However, our goal in **RQ4** is only to provide a simulation of the proportion of crash occurrences that can be avoided.

Internal validity threats concern factors that may affect a dependent variable and were not considered in the study. In **RQ3**, we imposed a minimum size to our testing sets, *i.e.*, we considered releases with at least 25% of new bugs. However, to avoid biasing our results with this threshold, we performed additional evaluations of our proposed models using respectively 25%, 30%, and 35% of new bugs. In Bugzilla, all time stamps are reported in UTC timezone. Therefore, reported *week day*, *month day*, and *month* might not precisely reflect developers' local time. However, from all these metrics, only *month* contributed significantly to the models (see the results about top predictors). This metric (*i.e.*, *month*) is less likely to be biased by time zone conversions.

Conclusion validity threats concern the relation between the treatment and the outcome. We paid attention not to violate assumptions of the performed statistical tests. To determine the cut-off of bugs with high crash entropy and high crash frequency, we conducted a sensitivity analysis. We applied different threshold of 50%, 70%, and 90% of percentiles to verify the characteristics of highly-impactful bugs. Results show that different percentiles do not affect the conclusion. For any detail about the sensitivity analysis, please check our data at: `https://github.com/swatlab/highly-impactful`. We used non-parametric tests that do not require making assumptions about the data set distribution. In **RQ2**, we did not investigate the characteristics of the four categories of bugs with respect to priority and severity assigned in Bugzilla because, in our previous work [4], we found that the priority and severity labels in Mozilla's bug reports do not reflect the concrete levels of attention paid by developers when fixing the bugs.

External validity threats concern the possibility to generalise our results. Although we only conduct our case study with two Mozilla subsystems, because only the Mozilla Foundation has opened their crash collecting database to the public [17] to date, most of our findings are consistent with previous studies [4, 12]. We share our data and scripts at: `https://github.com/swatlab/highly-impactful`. Further studies with different systems are required to verify our results and make our findings more generic.

4.6 Chapter Summary

Bug triaging guides software practitioners to focus their effort to address bugs with high priority when resources are limited. Current bug triaging approaches only take bugs' crash frequency into account while ignoring the impact on end users. Although previous studies used entropy analysis to improve the current bug triaging approaches, these approaches were applied only after end users have already suffered crashes for a certain period of time. In this chapter, after examining the prevalence and characteristics of highly-impactful bugs, *i.e.*, bugs with high crashing frequency and entropy, in Mozilla Firefox and Fennec, we built predictive models to help software organisations predict them early before they impact a large population of users. Our proposed models can predict highly-impactful bugs with a precision up to 64.2% (in Firefox) and a recall up to 98.3% (in Fennec). Using a simulation to evaluate the benefit of our best predictive model, cforest, we found that, on average, our early prediction technique can effectively prevent 23.0% of crash occurrences in Firefox (respectively 13.4% in Fennec) and reduce 28.6% of unique machine profiles that are impacted in Firefox (respectively 49.4% in Fennec). Software organisations could use our suggested predictive models to identify highly-impactful bugs and improve the satisfaction of their users. In the future, we plan to implement our approach in a tool and validate our results on different software systems. We also appeal to other software organisations to share their crash reports databases with the public to help generalise the results of our study.

CHAPTER 5 CRASH-INDUCING COMMITS

Software crashes are feared by software organisations and end users. In the previous chapter, we built statistical models in Mozilla projects to predict crash-related bugs that lead to frequent crashes, which impact a large user base. This improved approach can be applied at an early stage of development to detect crash-related bugs with a serious negative impact on users, but software development teams still have to wait for a certain period, during which crashes are collected, triaged and filed into bug reports, before they can carry out their bug fixing activities. If software organisations could detect crash-prone code even earlier, at the time of commits, they could address the problems as soon as possible and prevent the unpleasant experience of crashes to the users. This approach is referred to as "Just-in-Time Quality Assurance" [32], which enables fine-grained defect predictions and allows quality assurance teams to identify error-prone code early on. By identifying error-prone commits early, quality assurance teams are also likely to make better decisions choosing developers to fix a bug.

In this chapter, we investigate statistical models to predict commits that may introduce crashes (referred as "crash-inducing commits") in Mozilla Firefox. We are limited to Firefox because, at the time of this writing, no other organisation provides access to its crash reporting system. Software organisations can apply our proposed approach to detect crash-prone code early on before they affect a large number of users and address the defective code as soon as possible. We study Mozilla Firefox' crash reports between January 2012 and December 2012, as well as its commit logs from the beginning of the project until December 2012, and answer the following research questions:

RQ1: *What is the proportion of crash-inducing commits in Firefox?*

> We analyse Firefox' crash reports and link them to the corresponding crash-related bugs. We then use the SZZ algorithm [39] to map these bugs to their related commits and identify the commits due to which the crash-related bugs occurred. We found that crash-inducing commits account for 25.5% in the studied version control system.

RQ2: *What characteristics do crash-inducing commits possess?*

> By investigating the characteristics of crash-inducing commits and other commits, we found that, in general, crash-inducing commits are submitted by developers with less experience and are more often committed by developers from Mozilla. Developers change more files, add and delete more lines in crash-inducing commits. Compared to

other commits, more crash-inducing commits fix a previous bug, and often, they lead to another bug. In terms of changed types, crash-inducing commits contain more unique changed types and the changed statements tend to be scattered in more changed types, while other commits tend to be changed on a specific changed type.

RQ3: *How well can we predict crash-inducing commits?*

Previous studies, which proposed statistical models to predict defects from bug reports, could be effective to some extent. However, before a certain type of crashes is filed into the crash collecting system, a large number of end users might have already suffered a negative experience. Moreover, during this period, developers may become less familiar with the code. In this case, they may spend more time identifying the erroneous lines to fix the problems. Therefore, statistical models that can predict error-prone code just-in-time are required to help software practitioners detect crash-inducing commits and effectively fix them early. We use GLM, Naive Bayes, C5.0, and Random Forest algorithms to predict whether or not a commit will induce future crashes. Our predictive models can reach a precision of 61.4% and a recall of 95.0%. Software organisations can apply our proposed technique to improve their defect triaging process and the satisfaction of their users.

Chapter Overview

Section 5.1 explains the identification technique of crash-inducing commits. Section 5.2 describes data collection and processing for the empirical study. Section 5.3 presents and discusses the results of the three research questions. Section 5.4 discusses threats to the validity. Section 5.5 summarises this chapter.

5.1 Identification of Crash-inducing Commits

In this section, we describe the identification procedure for crash-inducing commits. All of our data and analytic scripts are available at:
`https://github.com/swatlab/crash-inducing`.

Applying the SZZ algorithm [39], we identify crash-inducing commits in two steps: identification of crash-related bugs and identification of commits that induce those bugs. The remainder of this section elaborates on each of these steps.

5.1.1 Identification of Crash-related Bugs

We extract the bug list from each of the studied crash reports. For each of the crash-related bug, we use regular expressions to identify the crashed stack trace from the bug's title and comments, then extract crash-related files or methods from the stack trace. We record the identified files or methods as defective locations of the crash-related bugs, which will be used to identify crash-inducing commits in the next step. Each crash-related bug may be linked to multiple crash occurrences. We sort these crashes by time and record the dates of the first and the last crash occurrences before the bug was opened.

5.1.2 Identification of Crash-inducing Commits

Since Śliwerski et al. [39] introduced the SZZ algorithm, a plethora of studies (such as [52, 53, 54]) have leveraged this approach to identify the commits that induce subsequent commits, especially bug fixes, in version control systems. In this paper, we use the SZZ algorithm to identify the commits that lead to crash-related bugs as follows.

Extraction of Crash-related Changed Files

We use heuristics proposed by Fischer et al. [37] to map the crash-related bug IDs to their corresponding bug fixes. We use regular expressions to detect bug IDs from the message of each commit. We then manually eliminate the false positives from the results. Some commits that fix previous bug fixes, *i.e.*, supplementary bug fixes, may lack bug identifier in the commit messages, where only a commit ID of a previous fix is provided. We track these commit IDs back to their original commits and check whether these original commits could be mapped to a bug report. Thus, we ensure that every crash-related bug can be map to all possible corresponding commits. As Mozilla's revision history is managed by Mercurial, for each of the identified bug fixes, we run a Mercurial command to extract its modified files and deleted files:

```
hg log --template {rev}, {file_mods}, {file_dels}
```

Here, we do not take added files into account, because only modified and deleted files could be changed by preceding commits.

Identification of the Previous Commits of the Changed Files

The changed files identified in Section 5.1.2 (*i.e.*, modified and deleted files) are considered as files that address the crash-related bugs. For each of the changed files in a certain commit C to

the bug B_{crash}, if its previous commit C' is dated before the bug's first crash occurrence date, C' would be considered as a "crash-inducing commit". Concretely, to seek out the previous commits of each changed file to a specific commit, we use Mercurial's `annotate` command to track the previous commit ID of each line in this file. Among the identified commit IDs, we first remove those related to white spaces and comment lines. The remaining commit IDs are candidates of crash-inducing commits. Then, for each of the IDs, we record its committed date as $D_{candidate}$. We find out the first crash date D_{first} of the bug B_{crash}, which is extracted in Section 5.1.1, and compare it with $D_{candidate}$. If $D_{candidate}$ is earlier than D_{first}, this candidate commit is identified as a "crash-inducing commit". Otherwise, if $D_{candidate}$ is later than D_{first}, but earlier than the last crash date D_{last} before the opening of the bug, we check whether this candidate commit contains any of the files appearing in the crashed stack trace of B_{crash}. If yes, we also include this commit into the set of crash-inducing commits.

All of the above steps have been implemented in Python scripts. Future researchers can use our scripts to validate our data analysis process or conduct their replication studies.

5.2 Case Study Design

This section describe the data collection and processing for our case study, which aims to address the following three research questions:

1. What is the proportion of crash-inducing commits in Firefox?
2. What characteristics do crash-inducing commits possess?
3. How well can we predict crash-inducing commits?

5.2.1 Data Collection

We analyse crash reports of Mozilla Firefox from January 2012 until December 2012. Since a crash-inducing commit cannot be submitted later than any of its related crashes, we select the revision history of Mozilla Firefox from the beginning of the project until December 2012. In summary, there are in total 132,484,824 crash reports (grouped into 2,210,126 crash-types) and 127,212 commits selected in this study.

5.2.2 Data Processing

Figure 5.1 shows an overview of our data processing steps for the case study. The corresponding data and Python scripts are available at: `https://github.com/swatlab/crash-inducing`.

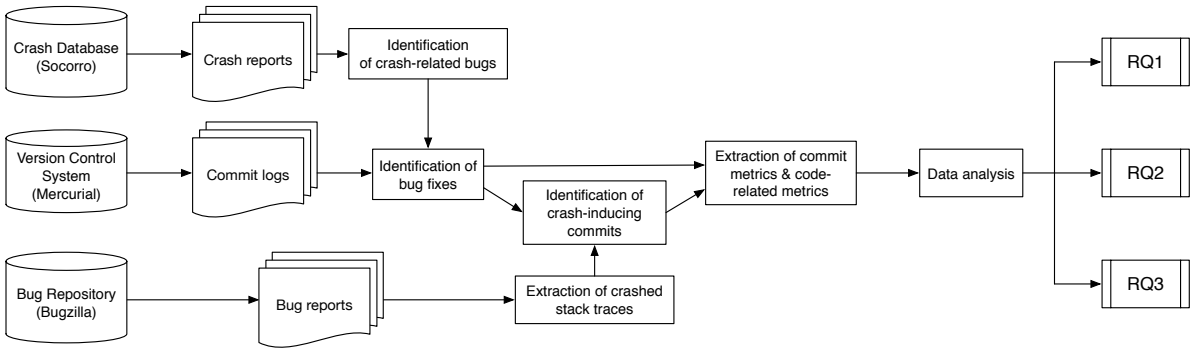

Figure 5.1 Overview of our approach to identify crash-inducing commits and extract their characteristic metrics

Mining Crash Reports

To identify crash-inducing commits and investigate the characteristics of these commits, we extract the following metrics from each crash reports: *bug list*, *crash date*, and *release number*. We use the bug IDs in the bug list to map a crash report to its bug reports. We then use crash date to find the earliest and the latest crash occurrence date before opening of each bug (see Section 5.1.1). We use the source code of all detected releases to compute code complexity metrics and social network analysis metrics.

Computing Code Complexity Metrics

For each studied commit, we use the Mercurial `log` command to extract all of its changed files. Then, as in Chapter 4, we apply the source code analysis tool *Understand* [47] to compute the code related metrics of the analysed files and identify the relationship among these files. We use a Python script to extract five metrics on code complexity for the files in each subject commit: lines of code (LOC), average cyclomatic complexity, number of functions, maximum nesting, and ratio of comment lines over all lines in a file. Because more than 90% of Firefox' code is written in C or C++ (see Chapter 4), in this step, we only take C and C++ files into consideration. Details of the selected code complexity metrics are discussed in Section 5.3.

Computing Social Network Analysis Metrics

From the the Understand database generated in Section 5.2.2, we identify the dependency among different files in Firefox and compute Social Network Analysis (SNA) metrics for each file. We compute the following social network analysis metrics: PageRank, betweenness, closeness, indegree, and outdegree. Details of the selected SNA metrics are discussed in

Section 5.3.

In Section 5.2.2 and Section 5.2.2, we compute the code related metrics for each of the releases detected from Section 5.2.2. For a given commit C whose committed date is D_c, we search the latest release R whose release date D_r is satisfied: $D_r < D_c$. We map all the files in the commit C to the release R and record the code complexity and SNA metrics for each of the successfully mapped files.

Identifying Changed Types

In a commit, different types of changes affect a software system to different extents in terms of crashes. For example, comment changes and refactoring would have little probability to trigger subsequent crashes. Yet, if parameters or function calls are not appropriately modified (or added/deleted) in a commit, crashes would probably happen when the commit is integrated into the version control system. We use the source code analysis tool *srcML* [55] to convert C or C++ code into XML files where each syntactic statement will be converted into an XML node, in which an XML tag labels its type. For a given changed file F in a certain commit C, we use the following Mercurial command to check it out:

```
hg cat -r C F
```

Then, we also check out the file with the same name F' in the previous commit C'. After converting F and F' into XML format, we use a Python script to recursively compare the difference on each of the corresponding srcML tags[1]. As we detected more than 80 unique srcML tags from the studied changed files, we group the srcML tags with similar semantic functions into a "changed type", while ignoring trivial srcML tags, such as "block" and "@format". Table 5.1 shows all of changed types and their corresponding srcML tags.

Besides counting the number of changed types in a commit, we also investigate the distribution of the changed types in the commit. We compute the value of the normalised Shannon entropy [45], defined as:

$$H_n(C) = -\sum_{i=1}^{n} p_i \times log_n(p_i) \tag{5.1}$$

where C is a commit; p_i is the probability of C possessing a specific changed type CT_i ($p_i \geq 0$, and $\sum_{i=1}^{n} p_i = 1$); n is the total number of unique changed types listed in Table 5.1. So, for a commit, if all changed types have the same occurrences, *i.e.*, the changed types are equally distributed, the entropy is maximal (*i.e.*, 1). If a commit only has one changed type, the entropy is minimal (*i.e.*, 0).

[1]For all srcML tags, please refer to:
`http://www.srcml.org/doc/srcMLGrammar.html`

Table 5.1 Changed types identified from Firefox' source code

Changed type	**srcML tag(s)**
Class	*class, class_decl, member_list*
Comment	*comment*
Constructor	*constructor, constructor_decl*
Control flow	*while, do, if, else, break, goto, for, foreach, continue, then, switch, case, return, condition, incr, default*
Data structure	*enum, struct, struct_decl, typedef, union, union_decl*
Declaration	*asm, decl, decl_stmt, using, namespace, range, specifier*
Destructor	*destructor, destructor_decl*
Function	*function, function_decl*
Init	*init*
Invocation	*call*
Access modifier	*super, public, private, protected, extern*
C++ feature	*template*
Parameter	*param, parameter_list, argument, argument_list*
Preprocessor	*cpp:define, cpp:elif, cpp:else, cpp:endif, cpp:error, cpp:file, cpp:if, cpp:ifdef, cpp:ifndef, cpp:include, cpp:line, cpp:pragma, cpp:undef, cpp:value, cpp:derective, macro*
Refactoring	*name, typename, label*
Variable Type	*type*

5.3 Case Study Results

This section presents and discusses the results of our three research questions. For each question, we discuss the motivation, the approach designed to answer the questions, and the findings.

RQ1: What is the proportion of crash-inducing commits in Firefox?

Motivation. This question is preliminary to the other questions. It provides quantitative data on the prevalence of commits that induce subsequent crashes in Mozilla Firefox. The results of this question will help software managers realise the prevalence of the crash-inducing commits and adjust their bug triaging strategy to focus the resources to resolve defects causing the crashes as soon as possible.

Approach. We identify crash-inducing commits using the technique presented in Section 5.1, then calculate their percentage over the total number of studied commits.

Finding. Among the 127,212 analysed commits, 32,463 are identified to result in future crashes. Figure 5.2 illustrates the proportion of crash-inducing commits and other commits (referred as crash-free commits hereafter).

Crash-inducing commits account for more than 25% of the total number of studied commits in Firefox.

One out of every four commits would cause subsequence crashes, which are considered as severe defects [56], because crashes can unexpectedly stop users' running process, lead to negative user experience and even decrease the reputation of a software organisation. Therefore, software practitioners should capture crash-inducing commits quickly, *i.e.*, when they are submitted into the version control system to address them as soon as possible. In the rest of this section, we will investigate the characteristics of crash-inducing commits and examine how to effectively predict them early.

RQ2: What characteristics do crash-inducing commits possess?

Motivation. Crash-inducing commits lead to bad user experience. If such a problem was not addressed promptly, developers would have to re-understand the code to locate the erroneous lines. Understanding the characteristics of crash-inducing commits can help software practitioners be aware of the factors that lead to crashes of a software system, and build predictive models to prevent them just-in-time.

Approach. For each of the commits identified either as crash-inducing commit or crash-free commit, we parse its commit log to extract the metrics presented in Table 5.2. We test the following nine null hypotheses to statistically compare the characteristics between crash-inducing commits and crash-free commits.

Comparing the extents of changes in crash-inducing vs. crash-free commits.

H_{01}^{1}: *the number of words is the same for crash-inducing and crash-free commits.*

H_{01}^{2}: *the number of changed files is the same for crash-inducing and crash-free commits.*

H_{01}^{3}: *the number of added lines is the same for crash-inducing and crash-free commits.*

H_{01}^{4}: *the number of deleted lines is the same for crash-inducing and crash-free commits.*

Comparing the changed types of crash-inducing vs. crash-free commits.

H_{02}^{1}: *the number of unique changed types is the same for crash-inducing and crash-free commits.*

H_{02}^{2}: *the entropy value of changed types is the same for crash-inducing and crash-free commits.*

Comparing the people and bug related factors of crash-inducing vs. crash-free commits.

H_{03}^{1}: *committers' experience is the same for crash-inducing and crash-free commits.*

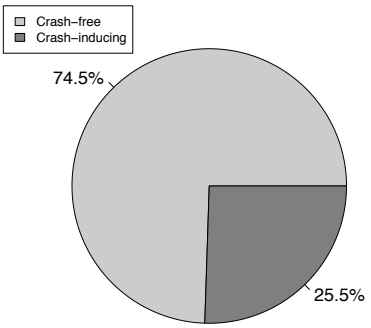

Figure 5.2 Proportion of crash-inducing commits and crash-free commits in Firefox

Table 5.2 Metrics used to compare the characteristics between crash-inducing commits and crash-free commits

Metric	**Description and rationale**
Committer's experience	Number of prior submitted commits.
Message size	Number of words in a commit message.
Changed files	Number of changed files (including added, deleted, and modified files) in a commit.
Added lines	Number of added lines of code in a commit.
Deleted lines	Number of deleted lines of code in a commit.
Number of changed types	Number of unique changed types in a commit.
Entropy of changed types	Measurement of the dispersion of different changed types in a commit (see Section 5.2.2).
Using Mozilla email	Whether a committer uses a Mozilla email address.
Is bug fix	Whether a commit is aimed to fix a bug.

H_{03}^{2}: *the percentage of Mozilla committers is the same for crash-inducing and crash-free commits.*

H_{03}^{3}: *the percentage of bug fixing commits is the same for crash-inducing and crash-free commits.*

We use the Wilcoxon rank sum test [49] to accept or reject the 7 first null hypotheses. As for H_{03}^{2} and H_{03}^{3}, we simply compare the percentage values between crash-inducing commits and crash-free commits. We use a 95% confidence level (*i.e.*, p-value < 0.05) to decide whether to reject a null hypothesis. Because we will conduct 7 null hypothesis tests, to counteract the problem of multiple comparisons, we apply the Bonferroni correction [50] that consists in dividing the threshold p-value by the number of tests. Thus, our threshold to decide whether a result is statistically significant is p-value $< 0.05/7 = 0.007$.

Table 5.3 Median value of characteristic metrics for crash-inducing commits and crash-free commits, as well as the p-value of the Wilcoxon rank sum test

Metric	Crash-inducing	Crash-free	p-value
Committer's experience	190	246	<2.2e-16
Message size	12	11	<2.2e-16
Changed files	3	2	<2.2e-16
Added lines	9	5	<2.2e-16
Deleted lines	34	13	<2.2e-16
Number of changed types	3	2	<2.2e-16
Entropy of changed types	0.339	0.23	<2.2e-16
Using Mozilla email	41.8%	36.7%	–
Is bug fix	91.4%	83.5%	–

Finding. Table 5.3 shows the median values of crash-inducing commits and crash-free commits on the metrics listed in Table 5.2, as well as the p-value of the Wilcoxon rank sum test. According to the results, crash-inducing commits are submitted by developers with less experience, suggesting that novice developers tend to write error-prone code. The message size of crash-inducing commits is significantly longer than crash-free commits. It is possible that crash-inducing commits are more complex and hence developers need longer comments to describe these changes. In crash-inducing commits, developers change significantly more files, and add and delete more lines than crash-free commits. This result is consistent with previous studies [57, 58] where researchers found that relative code churn measures can indicate defect modules. In terms of changed types, crash-inducing commits possess more unique changed types and their changed types' entropy is higher than crash-free commits. The changed statements are distributed in more changed types in crash-inducing commits than in crash-free commits. This observation suggests that it is preferable to make semantically coherent changes (*i.e.*, changes of the same type) in commits. When developers modify the code with a lot of changed types (with the modifications equally distributed across the changed types), these modifications have a higher probability to induce crashes.

Another interesting finding is the fact that crash-inducing commits were mostly submitted by developers using Mozilla email accounts. This situation may be due to the fact that commits from outside contributors receive more scrutiny (through code review sessions) than those from Mozilla developers. Finally, most of our studied commits (either crash-inducing or crash-free) are bug fixing attempts. This finding confirms that bug fixing has become the major activity in software development [1]. A higher proportion of crash-inducing commits are aimed to fix bugs; meaning that modifying code to fix an existing bug is a risky task that can induce other bugs; confirming arguments from previous studies, such as [59], that legacy

code becomes difficult to maintain.

In light of results from Table 5.3, we reject null hypotheses $H_{01}^{1} \sim H_{01}^{4}$, $H_{02}^{1} \sim H_{02}^{2}$, and H_{03}^{1}. In other words, for all metrics listed in Table 5.2, there exists a statistically significant difference between crash-inducing commits and crash-free commits.

In general, crash-inducing commits are submitted by less experienced developers. They contain longer commit messages, more changed files and changed lines than crash-free commits. Crash-inducing commits contain more changed types, their changed statements tend to be scattered in different changed types. More crash-inducing commits are aimed to fix previous bugs. And more crash-inducing commits are submitted by developers using Mozilla email accounts (i.e., Mozilla developers).

RQ3: How well can we predict crash-inducing commits?

Motivation. Crash-inducing commits may negatively impact users' experience, decrease the overall software quality and even the reputation of the software organisation. If we can predict these defective commits early on, we will increase users' satisfaction and shorten the period between the introduction of these crash-related bugs in the system and their detection and correction. In fact, if the detection of a bug is done long time after its introduction in the system, developers are likely to have a hard time identifying the root cause of the bug because their knowledge of the code tends to decrease overtime. Hence, a delayed detection of bugs is likely to augment maintenance overhead. In Chapter 4, we extracted metrics from bug reports to predict highly impactful crash-related bugs. Although this approach can shorten bug triaging time to some extent, developers still have to wait for a certain period, during which crashes are collected, triaged and filed into bug reports, before they can carry out their bug fixing activities. During this period, end users (possibly in large numbers) may have suffered crashes of the software. A Just-in-Time detection of crash-inducing commits will enable developers to act immediately on crash-prone commits before they can negatively impact users.

Approach. We extract 24 metrics along four dimensions from respectively the studied commit logs and the corresponding source code of Firefox. Table 5.4 to Table 5.7 show our selected metrics (*i.e.*, independent variables for the prediction models) and their rationales.

To predict whether or not a commit will cause subsequent crashes, we apply the four following algorithms: General Linear Model (GLM), decision tree, Random Forest, and Naive Bayes. The former three were used in previous chapters. Naive Bayes are a set of logistic regression algorithms based on Bayes' theorem with strong independence assumptions between the features. Although independence is normally a poor assumption, in practice, this algorithm

Table 5.4 Commit log metrics

Attribute	**Explanation and Rationale**
Hour	Hour (0-24). Code committed at certain hours may lead to crashes (e.g., hours around quitting time).
Week day	Day of week (from Mon to Sun). Code committed on certain week days may be less carefully written (*e.g.*, Friday) [39, 40], and would lead to crashes.
Month day	Day in month (1-31). Code committed on certain days may be less carefully written (*e.g.*, before and during public holidays); resulting into subsequent crashes.
Month	Month of year (1-12). Code committed in some seasons may be less carefully written; resulting into crashes. (*e.g.*, December, during Chrismas holidays).
Day of year*	Day of year (1-366). Combined the rationales of month day and month.
Message Size	Number of words in a commit message. In RQ2, we found that crash-inducing commits are correlated with longer commit messages.
Experience	Number of prior submitted commits. In RQ2, we found that crash-inducing commits tend to be submitted by less experienced developers.
From Mozilla	Whether a committer uses a Mozilla email address. In RQ2, we found that crash-inducing commits are submitted often by Mozilla's developers.
Number of changed files	Number of changed files in a commit. In RQ2, we found that commits with more changed files tend to cause subsequent crashes.
Is bug fix	Whether a commit aimed to fix a bug. In RQ2, we found that crash-inducing commits are correlated with bug fixing code.
Is supplementary fix	Whether a commit is to fix a prior fixed bug. Supplementary fixes may enhance previous fixes and may be less likely to cause crashes.
Before crashed files	Percentage of a commit's files that caused crashes in prior commits. Crashed code may be difficult to fix, and still lead to future crashes.

Table 5.5 Code complexity metrics

Attribute	**Explanation and Rationale**
LOC	Median lines of code in all classes in a commit. In RQ2, we found that crash-inducing commits have higher code churn (*i.e.*, added/deleted lines).
Number of functions	Median number of classes' functions in a commit. A huge class may be difficult to understand or modify, and lead to crashes.
Cyclomatic complexity	Median cyclomatic complexity of the functions in all classes in a commit. Complex code is hard to maintain and may cause crashes.
Max nesting*	Median maximum level of nested functions in all classes in a commit. A high level of nesting increases the conditional complexity and may increase the crashing probability.
Comment ratio	Median ratio of the lines of comments over the total lines of code in all classes in a commit. Codes with lower ratio of comments may not be easy to understand, and may result in crashes.

often performs well [60]. In this chapter, to enhance the performance of Random Forest, we build 100 trees, each of which are with 5 randomly selected metrics.

Table 5.6 Social network analysis metrics (other metrics in this dimension share the rationale as PageRank. We compute median value of each metric for all classes in a commit.)

Attribute	**Explanation and Rationale**
PageRank	Time fraction spent to "visit" a class in a random walk in the call graph. If an SNA metric of a class is high, this class may be triggered through multiple paths. An inappropriate change to the class may lead to malfunctions in the dependent classes; resulting into crashes.
Betweenness	Number of classes passing through a class among all shortest paths.
Closeness	Sum of lengths of the shortest call paths between a class and all other classes.
Indegree	Numbers of callers of a class.
Outdegree	Numbers of callees of a class.

Table 5.7 Changed type metrics

Attribute	**Explanation and Rationale**
Number of changed types	Number of unique changed types in a commit. In RQ2, we found that crash-inducing commits tend to contain more changed types.
Entropy of changed types	Distribution of changed types in a commit (see Section 5.2.2). In RQ2, we found that crash-inducing commits tend to have higher entropy of changed types.

To deal with collinearity in the data, before building the predictive models, we apply the Variance Inflation Factor (VIF) analysis to eliminate correlated metrics. We set the threshold to 5, *i.e.*, metrics with VIF values over this threshold are considered as correlated and will be removed from the predictive models. In Table 5.4 to Table 5.7, removed metrics are marked with *.

We use ten-fold cross validation [43] to compute the accuracy, precision, recall, and F-measure for crash-inducing commits and crash-free commits. Because crash-inducing commits and crash-free commits are imbalanced in our data set, we under-sample the majority class instances, *i.e.*, we randomly deleted instances from the data set of crash-free commits to make the data sets of crash-inducing commits and crash-free commits to have the same number of instances. We do this under-sampling only during the training phase. We rank the importance of the independent variables (prediction metrics) to identify the top predictors for the algorithm with the best prediction results.

Finding. Table 5.8 shows the median accuracy, precision, recall, and F-measure for the four algorithms used to predict whether a commit will cause crashes in Firefox. According to the results, our models can predict crash-inducing commits with a precision up to 61.4% and a recall up to 95.0%. Random Forest is the best prediction algorithm, which obtains the best F-measure when predicting either crash-inducing commits or crash-free commits. Among the

Table 5.8 Accuracy, precision, recall, and F-measure (in %) obtained from GLM, Naive Bayes, C5.0, and Random Forest to predict crash-inducing commits and crash-free commits

Metric	**GLM**	**Bayes**	**C5.0**	**Random Forest**
Accuracy	67.5	41.7	69.9	**73.3**
Crash-inducing precision	59.5	38.6	57.2	**61.4**
Crash-inducing recall	37.3	**95.0**	76.6	76.5
Crash-inducing F-measure	45.8	54.7	65.4	**68.1**
Crash-free precision	69.8	77.8	82.6	**83.8**
Crash-free recall	**84.8**	10.0	66.4	71.4
Crash-free F-measure	76.7	17.7	73.5	**77.4**

22 selected metrics, the SNA metric *closeness* is ranked as the most important predictor in all the 10 phases of the cross validation. This metric evaluates the degree of centrality of a class in the whole project. Our obtained result suggests that when many other classes depend on a class, a change to this (central) class is likely to induce crashes. Moreover, *message size*, *number of changed files*, *outdegree*, and *percentage of before crashed files* are ranked as the second important predictors; meaning that the length of comments in a commit, the number of changed files, the number of callees of classes modified by a commit, and the crashing history of files modified in a commit are good indicators of the risk of crashes related to the integration of a commit in the code repository.

Our predictive models can achieve a precision of 61.4%, and a recall of 95.0%. The Random Forest algorithm achieves the best prediction performance. Closeness is ranked as the best predictor in this algorithm. Software organisations can make use of the proposed predictive models to track crash-prone commits as soon as they are submitted for integration in the code repository, for example, during code review sessions.

5.4 Threats to Validity

In this section, we discuss the threats to validity of our study following the guidelines for case study research [44].

Construct validity threats concern the relation between theory and observation. In this research, the construct validity threats are mainly due to measurement errors. We used the source code of the previous release to a commit to compute complexity and SNA metrics. More specifically, for a given file F in a commit C, we found the previous release R of C, and computed the code complexity and SNA metrics of F in the context of the release R. Although the new commit C could slightly affect the values of these metrics, we observed that in most commits there is no noticeable change. Also, computing the metrics every time

a new commit is submitted would delay the detection of the crash-inducing commits (because the computation of the metrics takes some time). In this study, as a compromise, we use the files in the previous release to estimate a current commit's code complexity and SNA metrics. In the future, we will experiment with parallel algorithms to compute these metrics in real time.

Internal validity threats concern factors that may affect a dependent variable and were not considered in the study. In Section 5.1.2, although we removed all candidates of crash-inducing commits that only changed comments and–or white space lines, our "crash-inducing commits" may still contain some false positives. Concretely, in a fix of a crash-related bug, not all of the changes are aimed to address defects. Some lines may be added because of a refactoring or an addition of a new feature. These changes are hard to identify with an automatic approach. In our future work, we plan to manually examine a sample of the identified crash-inducing commits, and report its precision and recall.

Conclusion validity threats concern the relation between the treatment and the outcome. We paid attention not to violate the assumptions of the constructed statistical models. In RQ2, we used non-parametric tests, which do not require making assumptions about the distribution of the data set. When mapping crash-related bugs to their bug fixes, we manually checked false positives from the results. In addition, we manually grouped different srcML tags into changed types as shown in Table 5.1.

External validity threats concern the possibility to generalise our results. In this chapter, we analysed only Mozilla Firefox. Although many software organisations are using crash collecting systems, to the best of our knowledge, only Mozilla Foundation has shared its crash reports to the public [17]. In Chapter 4, we used another Mozilla project, Fennec for Android, as a subject system to study crash-related bugs. However, the code of Firefox and Fennec are both managed by a Mercurial release branch, in which, the two sub-systems share some common components; making it hard to separate the two systems at the level of commits. We look forward to generalise our proposed approach to more software systems. We share our data and scripts at `https://github.com/swatlab/crash-inducing`. Researchers and software practitioners can use these data and scripts to validate our results and replicate our technique to other systems.

5.5 Chapter Summary

Crashes, which are unexpected interruptions of a software system, are one of the major source of frustration for users. Frequent crashes of a software system can significantly decrease

user-perceived quality and even affect the overall reputation of a software organisation. To help software practitioners identify crash-prone code early on, we conduct a study of crash-inducing commits in Mozilla Firefox. We found that crash-inducing commits account for more than 25% of all studied commits. We also found that, compared to other commits, crash-inducing commits are often submitted by developers with less experience and that they contain longer comments, more changed files and changed lines, as well as more changed types.

To help software practitioners track and fix crash-inducing commits as soon as possible, we built predictive models using various regression and machine learning algorithms. These predictive models achieved a precision up to 61.4% and a recall up to 95.0%.

Software organisations can use our proposed predictive models to detect crash-prone code as soon as they are submitted for integration in the source code repository. They could then correct the code quickly to avoid users from experiencing the crashes. In the future, we plan to generalise our approach to other software systems and implement it into tools for different programming languages.

CHAPTER 6 CONCLUSION

In this chapter, we conclude the thesis and summarise our findings. In addition, we will discuss the limitations of our proposed approaches and the directions for future work.

6.1 Summary

Bugs are hated by users and software organisations. However, during software development and usages, software practitioners and end users always suffer from them. Today, software organisations make great efforts on bug fixing when they maintain software systems. To some extent, debugging strategies and techniques would decide the productivity of a development team and the overall user-perceived quality of the product. Bug tracking systems are now used by most software organisations, which archive the important defects detected from their software systems. Developers can discuss on the bugs and review the patches (bug fixes) through the bug tracking systems. Because bugs have different impacts on a software system and end users, software managers ought to assess the severity of the bugs and prioritise them to focus their limited time and resources to the most serious ones.

In this thesis, we used data analytics to study bug triaging techniques on three aspects: the relationship between supplementary bug fixes and re-opened bugs, an entropy-based bug triaging technique on crash-related bugs, and a Just-in-Time defect triaging technique on crash-inducing commits. We investigated the characteristics of different kinds of bugs and propose predictive models to help software organisations prevent them early on before they cause negative impact on their software and end users.

Relationship between Supplementary Bug Fixes and Re-opened Bugs

Previous studies show that bug re-opening can increase the maintenance costs as well as degrade the software quality and the satisfaction of users. To discover the relation between supplementary bug fixes and re-opened bugs, we investigate supplementary bug fixes where more than one fix are associated with the same bug and re-opened bugs in five open-source projects, and found that supplementary bug fixes account for 10.3% to 26.9% of total bug reports. In addition, in the subject systems, a high percentage (*i.e.*, from 21.6% to 33.8%) of the supplementary fixes have been re-opened. To help development teams target faulty/incomplete bug fixes (for more thorough reviews) and prevent re-opened bugs, we have explored the possibility of predicting bug re-openings over supplementary bug fixes, using GLM, C5.0, ctree, cforest and randomForest models. Results show that these models

can achieve between 72.2% and 97% precision as well as between 47.7% and 65.3% recall. Moreover, we found between 33.0% to 57.5% of re-opened bugs with only one commit associated to them. These re-opened bugs have a strong association with invalid bug reports in all our five studied systems. In fact, they were prematurely dismissed as "invalid" before being re-opened. These bugs are not as risky as re-opened bugs with more than one commit to the software development. In other words, contrary to claims by previous works on re-opened bugs, their impact to software quality is likely limited. The misclassification of these bugs reports may be due to developers' negative attitudes towards the bugs. Later, when developers are aware of the severity of these bugs, they must re-opened them again. Our proposed predictive models can help software managers improve their bug triaging process to prevent the misclassifications and the subsequent postponement of the software releases. Future researchers and practitioners mining data repositories can also use our models to identify fault-prone bug fixes.

Entropy-based Bug Triaging Technique on Crash-related Bugs

Bug triaging guides software practitioners to focus their effort to address bugs with high priority when resources are limited. Current bug triaging approaches only take bugs' crash frequency into account while ignoring the impact of bugs on end users. Although previous studies used entropy analyses to improve the current bug triaging approaches, these approaches were applied only after end users have already suffered crashes for a certain period of time. In this thesis, after examining the prevalence and characteristics of highly-impactful bugs, *i.e.*, bugs with high crashing frequency and entropy, in Mozilla Firefox and Fennec, we built predictive models to help software organisations predict them early before they impact a large population of users. Our proposed models can predict highly-impactful bugs with a precision up to 64.2% (in Firefox) and a recall up to 98.3% (in Fennec). Using a simulation to evaluate the benefit of our best predictive model, cforest, we found that, on average, our early prediction technique can effectively prevent 23.0% of crash occurrences in Firefox (respectively 13.4% in Fennec) and reduce 28.6% of unique machine profiles that are impacted in Firefox (respectively 49.4% in Fennec). Software organisations could use our suggested predictive models to identify highly-impactful bugs and improve the satisfaction of their users.

Just-in-Time Defect Triaging Technique on Crash-inducing Commits

To assist software organisations detect crashes even earlier, we studied crash-inducing commits in Mozilla Firefox. We found that one out of every four commits will induce future crashes. They are often committed by novice developers and contain longer comments, more changed files and changed lines, and more changed types. We built statistical models to pre-

dict crash-prone code at the level of commits to help software organisations fix the defective codes as soon as possible before they negatively impact end users. Our predictive models can achieve a precision of 61.4% and a recall of 95.0%. To enhance their bug triaging technique, software organisations could use our suggested models to capture crash-prone code just-in-time; preventing latent complaints from users.

6.2 Limitations of the proposed approaches

- We used a heuristic to automatically link a bug report to a commit where the bug's ID is detected. In certain bugs, developers may omit to write any bug identity in a bug fixing commit. On the other hand, a potential number series found in a commit may not correspond to a bug report, even though there exists a bug report identified by this number. We must eliminate false positives and false negatives by a manual analysis, which can help better understand the reasons of the supplementary fixes.
- In Chapter 4 and Chapter 5, we studied crash-related bugs merely on Mozilla projects. At the time of writing this thesis, too few software organisations have shared their crash collecting databases to the public. Although Mozilla Socorro is the only source that we can explore for a pertinent case study [17], replication studies on other subject software systems are required to validate our results and generalise our analytic techniques.

6.3 Future work

In the future, we plan to extend our study in the following directions:

- We will study the characteristics of invalid bug reports, *i.e.*, bugs that have been closed with the resolutions as invalid, wontfix, duplicate, and worksforme. Developers may prematurely close a serious bugs with these resolutions. To prevent the future reopening of the misclassified bugs reports, we will build predictive models on the invalid bug reports.
- We will generalise our entropy-based bug triaging technique and Just-in-Time crash detection technique on other software systems. We have contacted some software organisations to ask their crash reports for the future research. In addition, we also intend to implement our approach in a tool and validate our results on different software systems.

REFEFENCES

[1] N. I. of Standards & Technology, "The economic impacts of inadequate infrastructure for software testing," May 2002, uS Dept of Commerce.

[2] E. Shihab, A. Ihara, Y. Kamei, W. M. Ibrahim, M. Ohira, B. Adams, A. E. Hassan, and K.-i. Matsumoto, "Studying re-opened bugs in open source software," *Empirical Software Engineering*, vol. 18, no. 5, pp. 1005–1042, 2013.

[3] J. Park, M. Kim, B. Ray, and D.-H. Bae, "An empirical study of supplementary bug fixes," in *Mining Software Repositories (MSR), 2012 9th IEEE Working Conference on*. IEEE, 2012, pp. 40–49.

[4] F. Khomh, B. Chan, Y. Zou, and A. E. Hassan, "An entropy evaluation approach for triaging field crashes: A case study of mozilla firefox," in *Reverse Engineering (WCRE), 2011 18th Working Conference on*. IEEE, 2011, pp. 261–270.

[5] J. Anvik, L. Hiew, and G. C. Murphy, "Who should fix this bug?" in *Proceedings of the 28th international conference on Software engineering*, ser. ICSE '06. New York, NY, USA: ACM, 2006, pp. 361–370. [Online]. Available: http://doi.acm.org/10.1145/1134285.1134336

[6] G. Canfora and L. Cerulo, "Supporting change request assignment in open source development," in *Proceedings of the 2006 ACM symposium on Applied computing*, ser. SAC '06. New York, NY, USA: ACM, 2006, pp. 1767–1772. [Online]. Available: http://doi.acm.org/10.1145/1141277.1141693

[7] T. Menzies and A. Marcus, "Automated severity assessment of software defect reports," in *Software Maintenance, 2008. ICSM 2008. IEEE International Conference on*, 28 2008-oct. 4 2008, pp. 346 –355.

[8] C. Weiss, R. Premraj, T. Zimmermann, and A. Zeller, "How long will it take to fix this bug?" in *Proceedings of the Fourth International Workshop on Mining Software Repositories*, ser. MSR '07. Washington, DC, USA: IEEE Computer Society, 2007, pp. 1–. [Online]. Available: http://dx.doi.org/10.1109/MSR.2007.13

[9] G. Jeong, S. Kim, and T. Zimmermann, "Improving bug triage with bug tossing graphs," in *Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, ser. ESEC/FSE '09. New York, NY, USA: ACM, 2009, pp. 111–120. [Online]. Available: http://doi.acm.org/10.1145/1595696.1595715

[10] Z. Yin, D. Yuan, Y. Zhou, S. Pasupathy, and L. Bairavasundaram, "How do fixes become bugs?" in *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*. ACM, 2011, pp. 26–36.

[11] R. Purushothaman and D. E. Perry, "Toward understanding the rhetoric of small source code changes," *Software Engineering, IEEE Transactions on*, vol. 31, no. 6, pp. 511–526, 2005.

[12] T. Zimmermann, N. Nagappan, P. J. Guo, and B. Murphy, "Characterizing and predicting which bugs get reopened," in *Software Engineering (ICSE), 2012 34th International Conference on*. IEEE, 2012, pp. 1074–1083.

[13] X. Xia, D. Lo, X. Wang, X. Yang, S. Li, and J. Sun, "A comparative study of supervised learning algorithms for re-opened bug prediction," in *Software Maintenance and Reengineering (CSMR), 2013 17th European Conference on*, March 2013, pp. 331–334.

[14] Z. Gu, E. T. Barr, D. J. Hamilton, and Z. Su, "Has the bug really been fixed?" in *Software Engineering, 2010 ACM/IEEE 32nd International Conference on*, vol. 1. IEEE, 2010, pp. 55–64.

[15] M. Erfani Joorabchi, M. Mirzaaghaei, and A. Mesbah, "Works for me! characterizing non-reproducible bug reports," in *Proceedings of the 11th Working Conference on Mining Software Repositories*. ACM, 2014, pp. 62–71.

[16] A. Podgurski, D. Leon, P. Francis, W. Masri, M. Minch, J. Sun, and B. Wang, "Automated support for classifying software failure reports," in *Software Engineering, 2003. Proceedings. 25th International Conference on*. IEEE, 2003, pp. 465–475.

[17] S. Wang, F. Khomh, and Y. Zou, "Improving bug management using correlations in crash reports," *Empirical Software Engineering*, pp. 1–31, 2014.

[18] D. Kim, X. Wang, S. Kim, A. Zeller, S.-C. Cheung, and S. Park, "Which crashes should I fix first?: Predicting top crashes at an early stage to prioritize debugging efforts," *Software Engineering, IEEE Transactions on*, vol. 37, no. 3, pp. 430–447, 2011.

[19] "Socorro: Mozilla's crash reporting system," https://crash-stats.mozilla.com/home/products/Firefox, 2015, online; accessed June 13th, 2015.

[20] Y. Dang, R. Wu, H. Zhang, D. Zhang, and P. Nobel, "Rebucket: A method for clustering duplicate crash reports based on call stack similarity," in *Proceedings of the 2012 International Conference on Software Engineering*. IEEE Press, 2012, pp. 1084–1093.

[21] A. Bianchi, D. Caivano, F. Lanubile, and G. Visaggio, "Evaluating software degradation through entropy," in *IN ELEVENTH INTERNATIONAL SOFTWARE METRICS*

SYMPOSIUM, 2001, pp. 210–219.

[22] S. K. Abd-El-Hafiz, "Entropies as measures of software information," in *Proceedings of the IEEE International Conference on Software Maintenance (ICSM'01)*, ser. ICSM '01. Washington, DC, USA: IEEE Computer Society, 2001, pp. 110–. [Online]. Available: http://portal.acm.org/citation.cfm?id=846228.848671

[23] A. E. Hassan and R. C. Holt, "The chaos of software development," in *Proceedings of the 6th International Workshop on Principles of Software Evolution*. Washington, DC, USA: IEEE Computer Society, 2003, pp. 84–. [Online]. Available: http://portal.acm.org/citation.cfm?id=942803.943729

[24] S. Zaman, B. Adams, and A. E. Hassan, "Security versus performance bugs: A case study on firefox," in *Proceedings of the 8th Working Conference on Mining Software Repositories*, ser. MSR '11. IEEE Computer Society, 2011, pp. 93–102.

[25] K. Kim, Y. Shin, and C. Wu, "Complexity measures for object-oriented program based on the entropy," in *Proceedings of the Second Asia Pacific Software Engineering Conference*, ser. APSEC '95. Washington, DC, USA: IEEE Computer Society, 1995, pp. 127–. [Online]. Available: http://portal.acm.org/citation.cfm?id=785406.785448

[26] N. Chapin, "An entropy metric for software maintainability," *Twenty-Second Annual Hawaii International Conference on System Sciences, Software Track*, pp. 522–523, January 1995.

[27] E. Unger, L. Harn, and V. Kumar, "Entropy as a measure of database information," *Proceedings of the Sixth Annual Computer Security ApplicationsConference*, pp. 80–87, December 1990.

[28] J. Anvik, L. Hiew, and G. C. Murphy, "Coping with an open bug repository," in *Proceedings of the 2005 OOPSLA workshop on Eclipse technology eXchange*, ser. eclipse '05. New York, NY, USA: ACM, 2005, pp. 35–39. [Online]. Available: http://doi.acm.org/10.1145/1117696.1117704

[29] A. E. Hassan, "Mining software repositories to assist developers and support managers," in *Software Maintenance, 2006. ICSM'06. 22nd IEEE International Conference on.* IEEE, 2006, pp. 339–342.

[30] ——, "The road ahead for mining software repositories," in *Frontiers of Software Maintenance, 2008. FoSM 2008.* IEEE, 2008, pp. 48–57.

[31] A. E. Hassan and K. Zhang, "Using decision trees to predict the certification result of a build," in *Automated Software Engineering, 2006. ASE'06. 21st IEEE/ACM Interna-*

tional Conference on. IEEE, 2006, pp. 189–198.

[32] Y. Kamei, E. Shihab, B. Adams, A. E. Hassan, A. Mockus, A. Sinha, and N. Ubayashi, "A large-scale empirical study of just-in-time quality assurance," *Software Engineering, IEEE Transactions on*, vol. 39, no. 6, pp. 757–773, 2013.

[33] T. Fukushima, Y. Kamei, S. McIntosh, K. Yamashita, and N. Ubayashi, "An empirical study of just-in-time defect prediction using cross-project models," in *Proceedings of the 11th Working Conference on Mining Software Repositories*. ACM, 2014, pp. 172–181.

[34] A. T. Misirli, E. Shihab, and Y. Kamei, "Studying high impact fix-inducing changes," *Empirical Software Engineering*, pp. 1–37, 2015.

[35] T. Zimmermann, N. Nagappan, P. J. Guo, and B. Murphy, "Characterizing and predicting which bugs get reopened," in *Proceedings of the 2012 International Conference on Software Engineering*, ser. ICSE 2012. Piscataway, NJ, USA: IEEE Press, 2012, pp. 1074–1083. [Online]. Available: http://dl.acm.org/citation.cfm?id=2337223.2337363

[36] L. Breiman, "Random forests," *Machine learning*, vol. 45, no. 1, pp. 5–32, 2001.

[37] M. Fischer, M. Pinzger, and H. Gall, "Populating a release history database from version control and bug tracking systems," in *Software Maintenance, 2003. ICSM 2003. Proceedings. International Conference on*. IEEE, 2003, pp. 23–32.

[38] C. Bird, P. C. Rigby, E. T. Barr, D. J. Hamilton, D. M. German, and P. Devanbu, "The promises and perils of mining git," in *Proceedings of the 2009 6th IEEE International Working Conference on Mining Software Repositories*, ser. MSR '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 1–10. [Online]. Available: http://dx.doi.org/10.1109/MSR.2009.5069475

[39] J. Śliwerski, T. Zimmermann, and A. Zeller, "When do changes induce fixes?" in *ACM sigsoft software engineering notes*, vol. 30, no. 4. ACM, 2005, pp. 1–5.

[40] P. Anbalagan and M. Vouk, "Days of the week effect in predicting the time taken to fix defects," in *Proceedings of the 2nd International Workshop on Defects in Large Software Systems: Held in conjunction with the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2009)*. ACM, 2009, pp. 29–30.

[41] T. Mende and R. Koschke, "Effort-aware defect prediction models," in *Software Maintenance and Reengineering (CSMR), 2010 14th European Conference on*. IEEE, 2010, pp. 107–116.

[42] R. Díaz-Uriarte and S. A. De Andres, "Gene selection and classification of microarray data using random forest," *BMC bioinformatics*, vol. 7, no. 1, p. 3, 2006.

[43] B. Efron, "Estimating the error rate of a prediction rule: improvement on cross-validation," *Journal of the American Statistical Association*, vol. 78, no. 382, pp. 316–331, 1983.

[44] R. K. Yin, *Case Study Research: Design and Methods - Third Edition*, 3rd ed. SAGE Publications, 2002.

[45] C. E. Shannon, "A mathematical theory of communication," *SIGMOBILE Mob. Comput. Commun. Rev.*, vol. 5, pp. 3–55, January 2001. [Online]. Available: http://doi.acm.org/10.1145/584091.584093

[46] "SLOCCount," http://www.dwheeler.com/sloccount/, 2015, online; accessed June 13th, 2015.

[47] "Understand tool," https://scitools.com, 2015, online; accessed June 13th, 2015.

[48] "igraph," http://igraph.org/redirect.html, 2015, online; accessed June 13th, 2015.

[49] M. Hollander, D. A. Wolfe, and E. Chicken, *Nonparametric statistical methods*, 3rd ed. John Wiley & Sons, 2013.

[50] A. Dmitrienko, G. Molenberghs, C. Chuang-Stein, and W. Offen, *Analysis of Clinical Trials Using SAS: A Practical Guide*. SAS Institute, 2005. [Online]. Available: http://www.google.ca/books?id=G5ElnZDDm8gC

[51] F. Khomh, B. Adams, T. Dhaliwal, and Y. Zou, "Understanding the impact of rapid releases on software quality," *Empirical Software Engineering*, pp. 1–38, 2014.

[52] S. Kim, T. Zimmermann, K. Pan, and E. J. Whitehead Jr, "Automatic identification of bug-introducing changes," in *Automated Software Engineering, 2006. ASE'06. 21st IEEE/ACM International Conference on*. IEEE, 2006, pp. 81–90.

[53] B. A. Romo, A. Capiluppi, and T. Hall, "Filling the gaps of development logs and bug issue data," in *Proceedings of The International Symposium on Open Collaboration*. ACM, 2014, p. 8.

[54] C. Williams and J. Spacco, "SZZ revisited: verifying when changes induce fixes," in *Proceedings of the 2008 workshop on Defects in large software systems*. ACM, 2008, pp. 32–36.

[55] "srcML," http://www.srcml.org, 2015, online; accessed June 13th, 2015.

[56] R. Wu, "Diagnose crashing faults on production software," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 2014, pp. 771–774.

[57] R. Moser, W. Pedrycz, and G. Succi, "A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction," in *Software Engineering, 2008. ICSE'08. ACM/IEEE 30th International Conference on.* IEEE, 2008, pp. 181–190.

[58] N. Nagappan and T. Ball, "Use of relative code churn measures to predict system defect density," in *Software Engineering, 2005. ICSE 2005. Proceedings. 27th International Conference on.* IEEE, 2005, pp. 284–292.

[59] D. L. Parnas, "Software aging," in *Proceedings of the 16th international conference on Software engineering.* IEEE Computer Society Press, 1994, pp. 279–287.

[60] I. Rish, "An empirical study of the naive bayes classifier," in *IJCAI 2001 workshop on empirical methods in artificial intelligence*, vol. 3, no. 22. IBM New York, 2001, pp. 41–46.